VISIT…

Martin Bond
Dan Haywood
Debbie Law
Andy Longshaw
Peter Roxburgh

J2EE in 21 Days

SAMS
201 West 103rd St., Indianapolis, Indiana, 46290 USA

Sams Teach Yourself J2EE in 21 Days

Copyright © 2002 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32384-2

Library of Congress Catalog Card Number: 2001098579

Printed in the United States of America

First Printing: April, 2002

03 02 01 00 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Executive Editor
Michael Stephens

Managing Editor
Matt Purcell

Acquisitions Editor
Todd Green

Development Editor
Michael Watson

Project Editor
Christina Smith

Copy Editor
Pat Kinyon

Indexers
Tom Dinse
Erika Millen

Proofreader
Melissa Lynch

Technical Editor
Harold Finz, Steve Heckler, Farooq Karim, and Ari Krupnikov

Team Coordinator
Pamalee Nelson

Interior Designer
Gary Adair

Cover Designer
Aren Howell

Production
Cheryl Lynch
Michelle Mitchell

Contents at a Glance

	Introduction	1
WEEK 1	**Introducing J2EE and EJBs**	**7**
Day 1	The Challenge of N-Tier Development	9
2	The J2EE Platform and Roles	27
3	Naming and Directory Services	81
4	Introduction to EJBs	125
5	Session EJBs	165
6	Entity EJBs	211
7	CMP and EJB QL	271
WEEK 2	**Developing J2EE Applications**	**333**
Day 8	Transactions and Persistence	335
9	Java Message Service	395
10	Message-Driven Beans	429
11	JavaMail	461
12	Servlets	501
13	JavaServer Pages	555
14	JSP Tag Libraries	603
WEEK 3	**Integrating J2EE into the Enterprise**	**651**
Day 15	Security	653
16	Integrating XML with J2EE	701
17	Transforming XML Documents	741
18	Patterns	787
19	Integrating with External Resources	827
20	Using RPC-Style Web Services with J2EE	869
21	Web Service Registries and Message-Style Web Services	923
	Appendixes	
Appendix A	An Introduction to UML	965
B	SQL Reference	977
C	An Overview of XML	987
D	The Java Community Process	999
	Glossary	1003
	Index	1025

Contents

Introduction 1

WEEK 1 Introducing J2EE and EJBs 7

DAY 1 The Challenge of N-Tier Development 9

Monolithic Development 10
Consequences of Monolithic Applications 10
The Move into the Second Tier 11
Consequences of the 2-Tier Design 12
Complexity Simplified by Modularity 14
Component Technology 15
Benefits of Modularity 16
Benefits of the 3-Tier Scenario 16
A Model for Enterprise Computing 17
Lifecycle 18
Persistence 18
Naming 18
Transaction 19
Java 2 Enterprise Edition (J2EE) 20
Components and Containers 20
J2EE Standard Services 21
J2EE Blueprints 23
J2EE Compatibility Test Suite 24
The Future of J2EE 25
Summary 25
Q&A 25
Exercises 26

DAY 2 The J2EE Platform and Roles 27

Revisiting the J2EE Platform 28
Using Sun Microsystems' J2EE SDK 28
Installing J2EE SDK 1.3 29
Starting the J2EE Reference Implementation (RI) 32
Troubleshooting J2EE and Cloudscape 34
Closing Down J2EE RI and Cloudscape 37
Optional Software Used in this Book 37
Understanding Tiers and Components 38
The Business Tier 39
The Presentation Tier 44

Components: Web-Centric 45
The Client Tier 49
Standalone Client 52
Understanding Containers 55
Understanding the Services Containers Supply to Components 56
Hypertext Transfer Protocol (HTTP) 57
HTTP over Secure Sockets Layer (HTTPS) 57
Java Database Connectivity (JDBC) 57
Java Transaction API (JTA) 58
Java Authentication and Authorization Service (JAAS) 58
Java API for XML Parsing (JAXP) 58
Java Naming and Directory Interface (JNDI) 59
JavaBeans Activation Framework (JAF) 59
JavaMail 60
Java Message Service (JMS) 60
Java Interface Definition Language (Java IDL) 60
Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) 61
Connector Architecture 62
Introducing Platform Roles 62
J2EE Product Provider 63
Application Component Provider 63
Application Assembler 63
Application Deployer 64
Systems Administrator 64
Tool Provider 65
Future of J2EE tools 65
Packaging and Deploying J2EE Applications 66
J2EE applications 67
Breaking Modules down into Components 68
Summary 70
Q&A 70
Exercises—Case Study 71
The Job Agency 72
Using the Agency Case Study 73
Practice Makes Perfect 75
The Case Study Directory on the CD-ROM 76
Installing the Case Study Database 76

DAY 3 Naming and Directory Services 81

Naming and Directory Services 82
Why Use a Naming Service? 82
What is JNDI? 83

Common Naming Services 83
Naming Conventions 84
Using JNDI 85
Using Sun Microsystems' J2EE Reference Implementation 85
Obtaining an Initial Context 86
Initial Context Naming Exceptions 86
Defining the JNDI Service 87
JNDI Properties Files 88
Application Properties 89
Applet Parameters 90
Hard-Coded Properties 90
Binding JNDI Objects 90
Binding Objects 91
Binding Problems 91
Name Persistence 92
Rebinding Objects 92
Unbinding Objects 92
Renaming Objects 93
JNDI Name Lookup 93
Changing Contexts 94
Narrowing RMI-IIOP Objects 95
Contexts 96
Listing Contexts 96
Creating and Destroying Contexts 98
More on JNDI Names 100
Special Characters 100
Composite and Compound Names 100
URLs 101
Attributes 102
Overview of LDAP X.500 Names 102
Obtaining an LDAP Server 103
Using OpenLDAP 104
Configuring JNDI to use LDAP 106
Testing the LDAP Server 107
Obtaining a Directory Context 108
Reading Attributes 108
Searching for Objects 109
Manipulating Attributes 112
More on Objects 114
Loading Classes from a Code Base 114
Defining a Code Base 114
References 117

What Else Can JNDI Do? 120
JNDI Events 120
Security 121
Summary 122
Q&A 123
Exercise 124

DAY 4 Introduction to EJBs 125

What Is an EJB? 126
Beans, Clients, Containers, and Servers 126
The EJB Landscape 127
Discovering EJBs 127
Types of EJB 128
Common Uses of EJBs 128
Why Use EJBs? 129
Hiding Complexity 130
Separation of Business Logic from UI and Data Access 130
Container Services 131
What's in an EJB? 132
The Business Interface 132
The Business Logic 134
Factory Information 140
Bean Metadata 141
How Do I Create an EJB? 142
The Creation Mechanism 142
Caveats on Code Creation 143
Create the Deployable Component 143
How Do I Deploy an EJB? 147
Plugging into the Container 147
Performing the Deployment 148
How Do I Use an EJB? 148
Discovery 148
Retrieval and Use 149
Disposing of the EJB 150
Running the Client 150
Deploying and Using an EJB in the J2EE Reference Implementation 151
Opening the Case Study EAR File 152
Examining the Case Study Application 154
Deploying the Case Study Application 156
Testing the Case Study Application 158
Troubleshooting the Case Study Application 160

Summary 161
Q&A 161
Exercises 162

DAY 5 Session EJBs 165

Overview 165
The `javax.ejb` Package for Session Beans 167
Stateless Session Bean Lifecycle 168
Specifying a Stateless Session Bean 172
Implementing a Stateless Session Bean 175
Implementing `javax.ejb.SessionBean` 175
Implementing the Home Interface Methods 175
Implementing the Remote Interface Methods 177
Exceptions 179
Configuring and Deploying a Stateless Session Bean 180
Using `deploytool` 181
Structural Elements 182
Presentational Elements 183
Session Element 184
Deploying the Enterprise Application 193
Stateful Session Bean Lifecycle 193
Specifying a Stateful Session Bean 196
Implementing a Stateful Session Bean 198
Passivation 198
Timeouts 199
Chaining State 200
Configuring and Deploying a Stateful Session Bean 200
Client's View 201
Patterns and Idioms 202
Business Interface 203
Adapter 204
Coarse-Grained 205
Gotchas 205
Summary 206
Q&A 207
Exercises 207

DAY 6 Entity EJBs 211

Overview 211
The N-tier Architecture Revisited 212
Comparison with RDBMS Technology 213
Identifying Entities 214

The `javax.ejb` Package for Entity Beans 216
Entity Bean Types 217
Remote Versus Local Interfaces 217
BMP Entity Bean Lifecycle 219
Specifying a BMP Entity Bean 225
Local-Home Interface 225
Local Interface 230
Implementing a BMP Entity Bean 231
Implementing the Local-Home Interface Methods 235
Implementing the Local Interface Methods 241
Generating IDs 243
Granularity Revisited 245
Beware Those Finder Methods! 245
EJB Container Performance Tuning 247
Configuring and Deploying a BMP Entity Bean 248
Entity Element 249
Client's View 252
Session Beans Revisited 254
Patterns and Idioms 258
Interfaces, Façades, and State 258
Use Local Interfaces for Entity Beans 258
Dependent Value Classes 259
Self-Encapsulate Fields 261
Don't Use `Enumeration` for Finders 262
Acquire Late, Release Early 262
Business Interface Revisited 264
Gotchas 264
Summary 265
Q&A 266
Exercises 266

DAY 7 CMP and EJB QL 271

Overview of Container-Managed Persistence 271
N-tier Architecture (Revisited Again) and CMP Fields 273
A Quick Word about the Case Study Database 276
CMP Entity Bean Lifecycle 277
Container-Managed Relationships 279
Relationship Types 280
Navigability 282
`cmr-fields` 282
Manipulating Relationships 286

EJB QL 291
Select Methods 291
Syntax and Examples 293
Further Notes 300
Specifying a CMP Entity Bean 301
The Local-Home Interface 301
The Local Interface 301
Implementing a CMP Entity Bean 302
Implementing `javax.ejb.EntityBean` 302
Implementing the Local-Home Interface Methods 305
Finder Methods 308
Implementing the Local Interface Methods 312
Configuring a CMP Entity Bean 313
The `entity` Element 313
The `relationships` Element 317
Deploying a CMP Entity Bean 322
Patterns and Idioms 323
Normalize/Denormalize Data in `ejbLoad()/ejbStore()` 323
Don't Expose `cmp-fields` 324
Don't Expose `cmr-fields` 325
Enforce Referential Integrity Through the Bean's Interface 326
Use Select Methods to Implement Home Methods 327
Gotchas 328
Summary 329
Q&A 329
Exercises 330

WEEK 2 Developing J2EE Applications 333

DAY 8 Transactions and Persistence 335

Overview of Transactions 336
Container-Managed Transaction Demarcation 338
Bean Managed Transaction Demarcation 345
Motivation and Restrictions 345
Using the Java Transaction API 345
Deploying a BMTD Bean 349
Client-Demarcated Transactions 350
Exceptions Revisited 350
Extended Stateful Session Bean Lifecycle 352
Transactions: Behind the Scenes 354
Transaction Managers, Resource Managers, and 2PC 354
The JTA API 356

What If It Goes Wrong? 359
JTA Versus JTS 361
Overview of Persistence Technologies 363
JDBC 365
SQLj 367
SQLj Part 0 368
SQLj Part 1 373
SQLj Part 2 378
JDO 383
JDO Concepts 384
`javax.jdo` Classes and Interfaces 387
Queries 389
Other Features 391
Gotchas 392
Summary 393
Q&A 393
Exercises 394

DAY 9 Java Message Service 395

Messaging 395
Message Passing 396
Java Message Service API 397
JMS and J2EE 398
JMS API Architecture 399
Message Domains 400
Developing JMS Applications Using JBoss1 402
JMS Implementation in JBoss 402
Programming a JMS Application Using J2EE RI 404
J2EE RI Connection Factories 404
Adding Destinations in J2EE RI 404
Creating a Queue in J2EE RI 404
Point-to-Point Messaging Example 406
JMS Messages 407
Creating a Message 409
Sending a Message 409
Closing the Connection 410
Send JMS Text Message Example 410
Consuming Messages 411
Simple Synchronous Receiver Example 412
Receive JMS Text Message Example 413
Asynchronous Messaging 414
The Publish/Subscribe Message Domain 415

Point-to-Point Messaging Example 416
Bulletin Board Publisher 417
Bulletin Board Subscriber 418
Creating Durable Subscriptions 420
Additional JMS Features 422
Introduction to XML 425
What Is XML and Why Would You Use It? 425
Summary 426
Q&A 426
Exercise 427

DAY 10 Message-Driven Beans 429

What Are Message-Driven Beans? 430
The Message Producer's View 430
Similarities and Differences with Other EJBs 431
Programming Interfaces in a Message-Driven Bean 431
Life Cycle of a Message-Driven Bean 432
The Message-Driven Bean Context 433
Creating a Message-Driven Bean 434
Method-Ready Pool 434
The Demise of the Bean 435
Consuming Messages 435
Handling Exceptions 436
Container- and Bean-Managed Transactions 436
Message Acknowledgment 437
JMS Message Selectors 438
Writing a Simple Message-Driven Bean 439
Implementing the Interfaces 439
Running the Example 440
Creating the Queue 441
Deploying the Message-Driven Bean 442
Create a Sender Client to Create a Message 445
Developing the Agency Case Study Example 447
Step 1—Sender Helper Class 447
Step 2—Agency and Register Session Bean 449
Step 3—The Message-Driven Bean 451
Step 4—Create the JMS Queue 456
Step 5—Deploy the EJBS 456
Step 6—Testing the ApplicantMatch Bean 457
Using Other Architectures 457
Summary 458
Q&A 458
Exercise 458

DAY 11 JavaMail 461

Understanding E-Mail 462
SMTP 463
Post Office Protocol 3 (POP3) 463
Internet Message Access Protocol (IMAP) 464
Other Protocols 464
Multipurpose Internet Mail Extensions (MIME) 464
Introducing the JavaMail API 465
Setting up Your Development Environment 465
Sending a First E-mail 466
Creating a First E-mail 466
Creating Multi-Media E-mails 472
Creating the Message: Approach #1 472
Creating the Message: Approach #2 476
Sending E-mails with Attachments 482
Exploring the JavaMail API 485
Retrieving Messages 485
Deleting Messages 489
Getting Attachments 490
Authenticating Users and Security 494
Summary 497
Q&A 497
Exercises 499

DAY 12 Servlets 501

The Purpose and Use of Servlets 502
Tailored for Web Applications 502
Server and Platform Independence 503
Efficient and Scalable 503
Servlets Integration with the Server 503
Introduction to HTTP 504
HTTP Structure 504
Other HTTP Methods 507
Server Responses 507
Introduction to HTML 509
The Servlet Environment 513
Servlet Containers 513
The Servlet Class Hierarchy 513
Simple Servlet Example 514
Passing Parameter Data to a Servlet 519
How to Access Parameters 519
Servlet Example with Parameters 520

Using a POST Request 522
The Servlet Lifecycle 522
The Servlet Context 524
Web Applications 525
Web Application Files and Directory Structure 525
The Web Application Deployment Descriptor 526
Handling Errors 528
HTTP Errors 528
Servlet Exception Handling 529
Retaining Client and State Information 530
Using Session Objects 530
Hidden Form Fields 532
Cookies 532
Creating a Cookie 533
URL Rewriting 535
Servlet Filtering 535
Programming Filters 535
Example Auditing Filter 537
Deploying Filters 538
Event Listening 541
Deploying the Listener 543
Servlet Threads 545
Security and the Servlet Sandbox 546
Agency Case Study 546
`AgencyTable` Servlet Code 546
Deploying the `AgencyTable` Servlet 548
Summary 552
Q&A 553
Exercises 553

DAY 13 JavaServer Pages 555

What is a JSP? 556
Separating Roles 557
Translation and Execution 557
JSP Syntax and Structure 557
JSP Elements 558
First JSP example 560
JSP Problems 563
JSP Lifecycle 563
Detecting and Correcting JSP Errors 565
JSP Lifecycle Methods 569

JSP Directives 570
The `include` Directive 570
The `page` Directive 571
Accessing HTTP Servlet Variables 575
Using HTTP Request Parameters 576
Simplifying JSP pages with JavaBeans 577
What Is a JavaBean? 578
Defining a JavaBean 579
Getting Bean Properties 579
Setting Bean Properties 580
Initializing Beans 581
Using a Bean with the Agency Case Study 581
Adding a Web Interface to the Agency Case Study 585
Structure and Navigation 585
Look and Feel 588
Error Page Definition 595
Deploying the Case Study JSPs 597
Comparing JSP with Servlets 600
Summary 601
Q&A 601
Exercise 602

DAY 14 JSP Tag Libraries 603

The Role of Tag Libraries 604
Developing a Simple Custom Tag 605
Using a Simple Tag 605
The Tag Library Descriptor (TLD) 606
Custom Java Tags 608
The `doStartTag()` Method 610
The `"Hello World"` Custom Tag 611
Deploying a Tag Library Web Application 612
Defining the TLD Location 614
Using Simple Tags 614
Tags with Attributes 615
Tags that Define Script Variables 618
Iterative Tags 622
Co-operating Tags 626
Using Shared Scripting Variables 626
Hierarchical Tag Structures 627
Defining Tag Extra Info Objects 634
Validating Attributes 635
Defining Scripting Variables 637

Processing Tag Bodies 637
JavaServer Pages Standard Tag Library (JSPTL) 640
Using the JSPTL with the J2EE RI 641
Using the JSPTL `forEach` Tag 643
Other JSPTL Tags 645
JSPTL Scripting Language 645
Other Jakarta Tag Libraries 646
Summary 647
Q&A 647
Exercise 648

WEEK 3 Integrating J2EE into the Enterprise 651

DAY 15 Security 653

Security Overview 654
Security Terminology 654
Common Security Technology 656
Symmetric Encryption 656
Asymmetric Encryption 658
SSL and HTTPS 659
Checksums and Digests 660
Digital Certificates 660
Security in J2EE 661
J2EE Security Terminology 661
Working with J2EE RI Security 663
Security and EJBs 666
Defining EJB Security 666
Defining Roles 666
Defining the Security Identity 668
Defining Method Permissions 670
Mapping Principals to Roles 674
Using Roles as the Security Identity 676
Security in Web Applications and Components 682
Web Authentication 683
Configuring J2EE RI Basic Authentication 684
Declarative Web Authorization 685
Programmatic Web Authorization 691
Adding Programmatic Web Security to the Case Study 692
Using Secure Web Authentication Schemes 694
Security and JNDI 695
Simple LDAP Authentication 696
SASL Authentication 696

Summary 698
Q&A 699
Exercises 699

DAY 16 Integrating XML with J2EE 701

The Drive to Platform-Independent Data Exchange 702
Benefits and Characteristics of XML 703
Origins of XML 703
Structure and Syntax of XML 704
HTML and XML 705
Structure of an XML Document 705
Declarations 706
Elements 706
Well-Formed XML Documents 708
Attributes 708
Comments 709
Creating Valid XML 710
Document Type Definitions 710
Namespaces 714
Enforcing Document Structure with an XML Schema 715
How XML Is Used in J2EE 718
Parsing XML 718
The JAXP Packages 720
Parsing XML using SAX 720
Document Object Model (DOM) Parser 725
Modifying a DOM Tree 731
Java Architecture for XML Binding 732
Differences Between JAXP and JAXB 733
Extending the Agency Case Study 734
Step 1—Change Session Beans 735
Step 2—Amend the `MessageSender` Helper Class 736
Step 3—Amend the `ApplicantMatch` Message-Driven Bean 737
Summary 739
Q&A 739
Exercises 740

DAY 17 Transforming XML Documents 741

Presenting XML to Clients 742
Presenting XML to Browsers 743
Extensible Stylesheet Language (XSL) 744
XSL-FO XSL Formatting Objects 744

Extensible Stylesheet Transformations (XSLT) 745
Applying Stylesheets 746
Storing Transformed Documents on the Server 746
Presenting XML Documents and Stylesheets to the Client 747
Transforming the XML Document on the Server 747
Using XALAN with J2EE 748
Transforming XML Documents with XALAN 749
Using XALAN from the Command Line 750
Using XSLT in Java Applications 751
XSLT Stylesheets 755
Template Rules 756
Text Representation of XML Elements 761
Using XPath with XSLT 762
Default Stylesheet Rules 764
Processing Attributes 765
Using Stylesheet Elements 767
Processing Whitespace and Text 767
Adding Comments 769
Attribute Values 770
Creating and Copying Elements 771
Attributes and Attribute Sets 774
Additional XSL Elements 777
XSLT Compilers 780
Summary 781
Q&A 782
Exercises 782

DAY 18 Patterns 787

J2EE Patterns 788
What Are Patterns? 788
Why Use Patterns? 790
Types of Patterns 790
J2EE Patterns 791
Pattern Catalogs 792
Applying J2EE-Specific Patterns 792
Applying Patterns in a Context 793
Generic Patterns 794
J2EE Presentation-Tier Patterns 795
J2EE Business-Tier Patterns 795
J2EE Integration-Tier Patterns 796
Patterns Within J2EE 797

Patterns in Context 797
Analysing the Case Study 797
Session Facades and Entity EJBs 798
Data Exchange and Value Objects 800
Data Access Without Entity EJBs 804
Messages and Asynchronous Activation 811
Composing an Entity 812
Composing a JSP 813
JSPs and Separation of Concerns 817
Client-Side Proxies and Delegates 820
Locating Services 821
Any Other Business 822
Refactoring the Case Study 822
Directions for J2EE Patterns 823
Summary 824
Q & A 824
Exercises 825

DAY 19 Integrating with External Resources 827

Reviewing External Resources and Legacy Systems 828
Introducing Connector Architecture 829
Overview of the Architecture 829
Roles and Responsibilities 830
Using the Common Client Interface 834
Interacting with an EIS 834
Installing a Resource Adapter 835
Creating a First CCI Application 836
Managing Transactions and Exploring Records 843
Introducing Other Connectivity Technologies 848
Introducing CORBA 849
Introducing Java IDL 851
Using RMI over IIOP 851
RMI over JRMP Example 852
RMI over IIOP Example 857
Introducing JNI 860
Evaluation of Integration Technologies 865
Summary 865
Q&A 866
Exercises 867

DAY 20 Using RPC-Style Web Services with J2EE 869

Web Service Overview 870
What Is a Web Service? 870
Why Use Web Services? 872

Web Service Technologies and Protocols 873
Web Service Architecture 873
Web Services for J2EE 875
J2EE Web Service Architecture 875
Tools and Technologies 876
Integrating Web Services with Existing J2EE Components 878
Using an RPC-style SOAP-Based Web Service 879
RPC-Oriented Web Services 880
Setting up Axis under Tomcat 4.0 881
Service Description Information 883
Anatomy of a WSDL Document 883
Creating a Java Proxy from WSDL 885
Calling the Web Service Through SOAP 889
A Half-Way House 891
Debugging a SOAP Interaction 892
Implementing an RPC-Style SOAP-Based Web Service 894
Wrapping up a Java class as a Web Service 894
A Client for Your Web Service 898
Starting from WSDL 900
Using Axis JWS files 903
Session Context and Web Services 905
Wrapping Existing J2EE Functionality as Web Services 909
Parameter Types and Type Mapping 911
Mapping Between Java and SOAP/WSDL Types 911
Mapping Complex Types with Serializers 912
Going Further with Complex Type Mapping 919
Summary 919
Q&A 920
Exercises 921

DAY 21 Web Service Registries and Message-Style Web Services 923

Registries for Web Services 924
What is a Web Service Registry? 924
Why Do I Need One? 924
How Do They Work? 925
Types of Registry 925
ebXML Registry and Repository 926
UDDI Overview 928
Accessing Information in a UDDI Registry 929
Manipulating Service Information using UDDI4J 929
Manipulating Service Information Using the IBM WSTK Client API 932
Retrieving and Using Service Information 933

Using JAXR for Registry Access 934
A Generic Approach 934
Using JAXR to Store and Retrieve Service Information 936
Using a Message-Based SOAP Interface 937
Message-Style Versus RPC-style 937
Creating a Client 938
Creating a Service 939
Sending and Receiving SOAP Messages with JAXM 939
JAXM and J2EE 941
Configuring JAXM 941
Sending Basic SOAP Messages 942
Running the Simple Client 947
Populating the Message 947
Headers and Attachments 951
Receiving a SOAP Message Using JAXM 952
Using a JAXM Profile 955
Sending a Message Using a JAXM Profile 957
Receiving a Message Using a JAXM Profile 959
Summary 962
Q&A 962
Exercises 963

Appendix A An Introduction to UML 965

Introducing the UML 965
Use Case Diagrams 967
Class Diagrams 969
Associations 969
Attributes 970
Operations 971
Generalization 972
Constraints 973
Sequence Diagrams 973

Appendix B SQL Reference 977

Commonly Used SQL Statements (SQL99) 978
`ALTER TABLE` 978
`CREATE TABLE` 979
`CREATE VIEW` 979
`DELETE` 980
`DROP TABLE` 980
`DROP VIEW` 980
`INSERT` 980

SELECT 981
UPDATE 983
Commonly Used SQL Clauses 983
FROM 983
WHERE 983
GROUP BY 984
HAVING 984
ORDER BY 984

Appendix C An Overview of XML 987

What Is XML? 988
Elements 988
Declarations 989
Comments 990
Special Characters 990
Namespaces 991
Enforcing XML Document Structure 991
Document Type Definition (DTD) 992
XML Schema 995
Where to Find More Information 997

Appendix D The Java Community Process 999

Introducing the JCP 999
Getting Involved 1000
JCP Members 1000
Expert Groups 1000
The Public 1000
Process Management Office (PMO) 1001
Executive Committees 1001
Understanding the JSR Process 1001
Taking the Next Step 1002

Glossary 1003

Index 1025

About the Authors

The authors of this book work for Content Master Ltd., a technical authoring company in the United Kingdom specializing in the production of training and educational materials. For more information on Content Master, please see its Web site at `www.contentmaster.com`.

Martin Bond, B.Sc. M.Sc. C.Eng, M.B.C.S., was born near Manchester England in 1958. Martin left a budding academic career to develop parallel processing compilers for Inmos. Martin has designed and developed systems using C++, Java, and JavaScript and has developed training courses on Unix programming, Solaris security, Java programming, and XML. Martin has an honors degree and a masters degree in computer science from Aberystwyth, Wales, and is a European chartered engineer. Martin currently works as an IT trainer and consultant based in Cornwall, England.

Dan Haywood has been working on large and small software development projects for more than 12 years. These days, he fills his days with consulting, training and technical writing, specializing in OO design, Java and J2EE, Sybase technical consulting, and data modeling. Previously, Dan worked at Sybase Professional Services, performing a variety of roles, mostly in the financial industry, including architect, performance specialist, and project manager. Dan started his IT career at (what was then) Andersen Consulting, working as a developer on large-scale projects in government and in utilities. Dan is married and has a baby daughter.

Debbie Law B.Sc., was born in Romsey, England in 1959. Debbie started on compiler development for parallel processing systems, later working on the design and development of client server applications. As a technical manager for Siemens, she was one of a small group of select staff on an intensive learning program studying worldwide business practices, including several weeks at MIT and Harvard. Debbie has an honors degree in computer science from Southampton, England and currently works as an IT consultant based in Cornwall, England.

Andy Longshaw is a consultant, writer, and educator specializing in J2EE, XML, and Web-based technologies and components, particularly the design and architecture decisions required to use these technologies successfully. Andy has been explaining technology for most of the last decade as a trainer and in conference sessions. A wild rumor suggests that some people have managed to stay awake in these sessions. Despite being well educated and otherwise fairly normal, Andy still subjects himself and his family to “trial by unpredictability” by watching Manchester City FC far more often than is healthy.

Peter Roxburgh graduated with a first class degree with honors in business, and has since followed a diverse career path. From his home in the medieval walled town of Conwy, North Wales, he authors a wide-variety of training courses, and books including *Building .NET Applications for Mobile Devices* (Microsoft Press, 2002). He has also written and contributed to a number of journals and Web sites on cutting-edge technologies.

Peter spends his spare time playing guitar and bouldering on nearby sea cliffs and mountain crags. When he is not strumming or risking life and limb, he enjoys spending relaxing and quality time with his daughter, Chloe.

Dedication

To Sarah, for encouragement, advice, and regular supplies of flapjacks; and to Adam and Josh, for providing me with a life that doesn't revolve around computers. —AL

To Sue: Thank you for all these happy years. —Love, Dan.

Acknowledgments

The authors would like to thank the various project managers and editors involved in this book, without whom it would never have seen the light of day. Special thanks go to Suzanne Carlino at Content Master and Todd Green, Michael Watson, Christy Franklin, and the editing team at SAMS. We would also like to acknowledge the work of Alex Ferris and John Sharp in the initial phases of this project.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4770

E-mail: `feedback@samspublishing.com`

Mail: Michael Stephens
Executive Editor
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

The world has come a long way since Duke first started tumbling in early versions of Netscape Navigator. Java has outgrown its humble origins as a cool way of providing interactivity on Web pages and has found a new role as a major, server-side development platform. The actual Java language has changed little in the intervening years, but an enterprise-quality infrastructure has risen up around it. This infrastructure, Java 2 Enterprise Edition or J2EE for short, allows Java developers to create sophisticated and powerful enterprise applications that provide mission-critical functionality for many thousands of users.

Unlike competing platforms, such as Microsoft .NET, J2EE is a specification rather than a product. The capabilities and functionality of each release of J2EE is agreed on through the Java Community Process (JCP). The platform is then implemented by application server vendors and producers, such as BEA, IBM, iPlanet, ATG, SilverStream, and JBOSS. This means that J2EE developers have a choice of product vendors from whom to select, based on quality, support, or ease of use. The ability to submit technologies through the JCP, and the two-way flow that exists between the main Java vendors and the open-source community, ensures that a constant stream of new ideas helps to move J2EE forward.

This book intends to take you on a journey through the J2EE landscape, from the simplest components through design considerations and on to the latest Web Services. There is a lot to learn in three weeks—but this should provide the essential grounding you need to use the J2EE platform effectively. If you need to create robust enterprise applications and Java is your tool of choice, read on.

How This Book is Organized

Sams Teach Yourself J2EE in 21 Days covers version 1.3 of the J2EE platform. It is organized as three separate weeks that guide you through the different functionality provided by J2EE.

The first week gives you a broad grounding in J2EE before moving on to investigate Enterprise JavaBeans (EJBs) in detail:

- Day 1, “The Challenge of N-Tier Development,” defines the landscape in which J2EE applications operate and provides the architectural concepts with which you need to become familiar to create J2EE applications.

- Day 2, “The J2EE Platform and Roles,” takes you on a whistle-stop tour of the J2EE platform, the major technologies, the types of component from which J2EE applications are assembled, and the container with which they interact. You also install the J2EE platform and start to look at the case study used throughout the book.
- On Day 3, “Naming and Directory Services,” you start using your first J2EE API, the Java Naming and Directory Interface (JNDI), to store, retrieve, and manipulate information that can be accessed by all J2EE components.
- Day 4, “Introduction to EJBs,” introduces Enterprise JavaBeans (EJB)—the core technology of the J2EE platform. You will examine the role of EJBs and how they work. You will then deploy an example EJB and create a simple client application for it.
- On Day 5, “Session EJBs,” you will explore Session EJBs in more depth. This includes the creation of both stateful and stateless Session EJBs.
- Day 6, “Entity EJBs,” moves on to Entity EJBs and examines their role and lifecycle. Particular attention is paid to how state is stored and retrieved using Bean-Managed Persistence (BMP).
- On Day 7, “CMP and EJB QL,” the discussion of Entity EJBs expands to cover entities that use Container-Managed Persistence (CMP) to store and retrieve their state. This includes an exploration of the EJB Query Language and Container-Managed Relationships that were introduced in J2EE 1.3.

The second week moves beyond EJBs to look at asynchronous interaction and the development of Web-based components:

- On Day 8, “Transactions and Persistence,” you will delve deeper into the use of transactions in the J2EE platform—what they can achieve and how your components can take advantage of them. Some alternative persistence mechanisms are also explored.
- Day 9, “Java Message Service,” looks at asynchronous messaging with the Java Message Service (JMS) using message queues and topics. You will apply JMS to implement a producer and consumer of asynchronous messages.
- Day 10, “Message-Driven Beans,” builds on the coverage of JMS to associate message queues with Message-driven EJBs. You will create an EJB whose functionality is triggered on receipt of an asynchronous message.
- On Day 11, “JavaMail,” another asynchronous communication mechanism is examined—namely e-mail. You will learn how to send and retrieve e-mail under J2EE and how this can be applied to transport data in a J2EE application.

- Day 12, “Servlets,” is the first of three Web-oriented days that explore the creation of Web-oriented J2EE applications. This starts by creating servlets to take advantage of the EJB-based services you built earlier. You will look at the servlet lifecycle and central issues, such as session tracking and state management.
- Day 13, “JavaServer Pages,” looks at how JavaServer Pages (JSP) can help to integrate Java and J2EE functionality with HTML content. It examines the role of JSPs and how JavaBeans can be used to encapsulate Java functionality in JSPs.
- On Day 14, “JSP Tag Libraries,” you will use custom JSP tag libraries to encapsulate Java functionality to improve the maintainability of the JSP pages.

The third week explores essential aspects of enterprise applications, such as security and integration, before moving on to application design and ending with a look at the Web Service functionality that will form the future of J2EE:

- Day 15, “Security,” begins week 3 by applying security to your J2EE application. You will weigh up the benefits of declarative and programmatic security and how they can be applied within your application.
- On Day 16, “Integrating XML with J2EE,” you will examine the role of XML in J2EE applications. You will create J2EE components that produce and consume XML documents and process data using the Java APIs for XML Processing (JAXP).
- Day 17, “Transforming XML Documents,” focuses on the transformation of XML documents into other formats, including other dialects of XML, primarily using the XSLT transformation language. Again, JAXP allows you to do this programmatically from within J2EE components.
- On Day 18, “Patterns,” you will take some time to consider the bigger picture and examine design issues for J2EE applications. The specific focus will be on common patterns that have been found as people have applied J2EE technologies in live applications. You will use this knowledge to improve parts of the case study design.
- Day 19, “Integrating with External Resources,” explores the various technologies that can be used to integrate J2EE applications with non-J2EE components and services. These mechanisms include the Java Connector Architecture, CORBA, RMI-IIOP, and the Java Native Interface.
- Day 20, “Using RPC-Style Web Services with J2EE,” looks ahead to the use of J2EE components as Web Services. You will use common Web Service technologies (such as SOAP and WSDL) to expose Java functionality as Web Services, and look at how the Java API for XML-Based RPC (JAX-RPC) addresses this.

- Day 21, "Web Service Registries and Message-Style Web Services," concludes the examination of J2EE-based Web Services by examining the role of XML-based registries and how the Java API for XML Registries (JAXR) enables access to this information. You will also create a message-oriented producer and consumer of Web Services using the Java API for XML Messaging (JAXM).

About This Book

This book is a practical, down-to-earth guide for intermediate Java developers. It is not intended to be a reference book, with lists of API calls or extensive discussion of the inner workings of the technologies. Rather, it provides you with a grounding in applying the essential J2EE technologies and leads you through the essential steps required to get a program or component written, packaged, and deployed on the J2EE platform. By the time you finish *Sams Teach Yourself J2EE in 21 Days*, you should have the confidence to create or maintain code that uses any of the major J2EE APIs.

Who Should Read This Book?

This book in intended for experienced Java developers who have been involved with Java development for at least 3–6 months. You should be confident writing Java code and familiar with the commonly used Java 2 Standard Edition APIs, such as string handling, JDBC, collections, iterators, and so on.

In addition to a firm grasp of Java, the following knowledge will speed your progress through the book:

- An understanding of how the Web operates, such as the use of a Web browser to retrieve pages of HTML from Web Servers.
- Familiarity with XML syntax to the level of reading small extracts of XML containing elements, attributes, and namespaces.
- An understanding of relational databases and how data is structured in tables. A familiarity with basic SQL to the level of understanding simple queries, inserts, updates, and joins.
- Familiarity with distributed systems, such as n-tier development, client-server programming, and remote procedure calls.

If you are not familiar with one or more of these topics, don't panic! There are appendixes on the CD-ROM that provide introductory material on XML and SQL. The essential concepts of distributed systems and Web-based development are covered in the main body of the book as required.

How This Book is Structured

This book is intended to be read and absorbed over the course of three weeks. During each week, you read seven chapters that present concepts related to J2EE and the creation of enterprise applications in Java. Care has been taken to try to ensure that concepts and technologies are introduced in an appropriate order, so it is best to read the chapters sequentially if possible.

At the end of each lesson are a set of questions asked about the subject covered that day. Answers to these questions are provided by the authors. There are also exercises for you to test your newly found skills by creating some related application or service.

The exercises in the book are largely based around a case study that is described in detail at the end of Day 2. The files for the case study and worked solutions to the exercises can be found on the CD-ROM that accompanies this book. The idea of the case study is that it will help you apply J2EE technologies and techniques in a consistent context and as part of a working application. This should provide you with a deeper understanding of the technology involved and how to apply it than is possible working with standalone examples.

Typographic Conventions

A Note presents interesting, sometimes technical, pieces of information related to the surrounding discussion.

A Tip offers advice or suggests an easier way of doing something.

A Caution advises you of potential problems and helps you avoid causing serious damage.

Text that you type, text that should appear on your screen, and the names of Java classes or methods are presented in `monospace` type.

WEEK 1

Introducing J2EE and EJBs

1 The Challenge of N-Tier Development

2 J2EE Platform and Roles

3 Naming and Directory Services

4 Introduction to EJBs

5 Session EJBs

6 Entity EJBs

7 CMP and EJB QL

WEEK 1

DAY 1

The Challenge of N-Tier Development

The current trend in enterprise program development is to provide n-tier frameworks aimed at delivering applications that are secure, scalable, and available. To this end, Sun Microsystems introduced Java 2 Enterprise Edition (J2EE), and Microsoft Corporation ventured the .NET framework to help developers build applications that are Web-friendly and frequently used to deliver e-commerce solutions. There are a myriad of application servers available to house enterprise applications, and many service providers are writing modular tools to plug in and extend the rich functionality. The clients that are taking advantage of this distributed architecture can be as simple as a Web browser (a so-called *thin* client).

This is the overarching vision and the state of the art. But, how did we get here?

To understand this landscape, this chapter investigates the principles of multiple tiers, component environments, and standards that underlie the frameworks. One of the objectives will be to give you a clear understanding of concepts and

terminology used when discussing such frameworks. Such terminology can frequently be confusing and inconsistently used. As a start along this road, please note that for the purposes of the following discussions, a tier refers to a physical separation (a different machine), and a layer refers to a logical layer in software terms, such that multiple layers can be on the same machine.

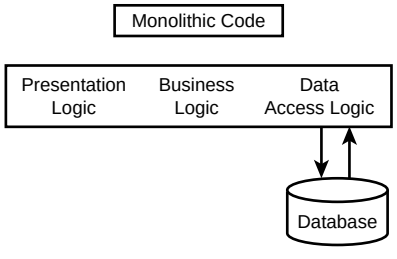
Monolithic Development

In the days of the mainframe or the standalone personal computer, when an application was housed on a single machine, it was common to find monolithic applications containing all the functionality of the application in one large, frequently unmaintainable piece of software (sometimes referred to as *spaghetti code*). All user input, verification, business logic, and data access could be found together. This suited the world of the mainframe and corporate data center because everything was controlled and the systems themselves tended to evolve slowly. However, as the world has speeded up over the last two decades, the high levels of maintenance required to keep up with changing business needs using such an application would mean that recompilation would be almost a daily event.

Even today, if you need a very simple application where, for example, the client application accesses and updates information on a database, locally you need only one tier. However, as you will see, you will still probably want to use components and or layers to control its complexity.

Figure 1.1 shows how this application may look running on a single machine.

FIGURE 1.1
Monolithic code scenario.



Consequences of Monolithic Applications

If you are writing a simple utility that does not use network connectivity, the previous scenario might suffice. However, any changes required to any part of the functionality may

potentially affect other parts. Because the Presentation, Business, and Data Access logic are located within the same piece of application code, recompilation of many parts of the code may be necessary, increasing the overhead of adding or changing functionality. Worse still, changes in part of the code may introduce unintentional bugs in other, seemingly unrelated, parts.

Of course, updating the application involves only one machine, but the rollout of new versions of the software gets more complicated as more users install and use the application.

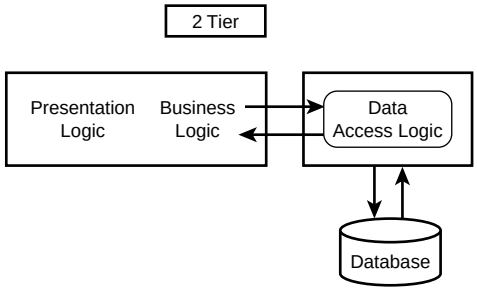
The Move into the Second Tier

The move towards 2-tier systems was born from the desire to share data between multiple applications installed on different machines. To do this, a separate database server machine was required. Figure 1.2 shows how this is achieved. The application now consists of presentation and business logic. Data is accessed by connecting to a database on another machine. Any changes to the Data Access logic should not affect the Presentation or Business logic in the application.

As indicated by Figure 1.2, splitting out Data Access Logic into a second tier keeps the data access independent and can deliver a certain amount of scalability and flexibility within the system.

FIGURE 1.2

2-tier scenario.



The advantage of having the Data Access Logic split into a separate physical environment means that not only can data be shared, but any changes to the data access logic are localized in that second tier. In fact, the whole of the second tier could be replaced with a different database and different code as long as the interface between the two tiers remained the same.

This provides an alternative way of looking at the program logic. Each part of the logic from the monolithic system could be regarded as a separate layer.

The logical division into layers of functionality can be based on the different responsibilities of parts of the code, namely,

- *Presentation Logic*—This dictates how the user interacts with the application and how information is presented.
- *Business Logic*—This houses the core of the application, namely the rules governing the business process (or any other functionality) embedded in the application.
- *Data Access Logic*—This governs the connection to any datasources used by the application (typically databases) and the provision of data from those datasources to the business logic.

So, we have two tiers in Figure 1.2 with two logical layers. The Presentation and Business Logic layers are still lumped together as one piece of potentially monolithic code.

Consequences of the 2-Tier Design

One of the central problems faced by application developers using the type of architecture shown in Figure 1.2 was that the client is still full of business code and it still needs to know details about the location of its data sources. Because there is such a concentration of functionality on the client, this type of client is generally termed a *thick* client. Thick clients usually need to be updated whenever the application changes.

Because the users of a thick client application have much of the application code installed on their local systems, there is a need to install fresh copies of the updated application when changes are made. This presents a serious manageability issue in terms of roll out and version control. Also, it is not always practical to use a thick client, because the application user may not want to install code on his or her machine to use a particular application. Similarly, the application provider may not want to provide code containing its business logic to relatively unknown third parties, even if it is precompiled.

Another issue with the use of thick clients relates to data access. The need to provide access to the back-end data for all clients of the application severely limits the reach and scalability of the application.

In addition to these inherent problems, many applications written with tools aimed at the two-tier environment still had all of their code in a single executable module. This increased maintenance headaches because there was a need to update the program design and implementation if any changes are required to any part of the system. With the advent of the Internet, there was a movement to separate Business logic from the user interface. Internet users, or more precisely Web users, need to access applications

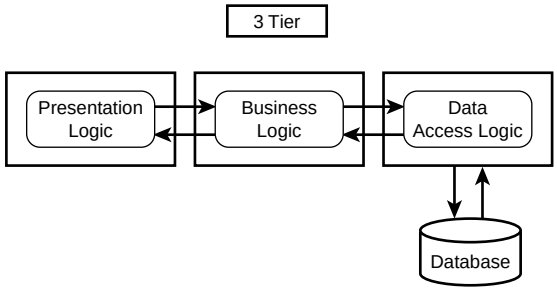
without installing new code on their machines. In fact, they want to be able to use the same client application—a Web browser—to access all of the different applications they encounter on the Web. Because the application logic associated with a thick client is no longer resident on the user's machine, this type of client is known as a *thin* client. The implication is that all of the "bulk" of the application has been moved into another tier. When a Web browser is used as a thin client, the application code will be run on the Web servers with which the browser communicates (or on other machines with which the Web servers communicate). The presentation tier logic for such an application must generate Hypertext Markup Language (HTML) rather than manipulate graphical elements on a GUI screen.

All of this has a serious implication for 2-tier systems. If a 2-tier system is to be adapted for use on the Internet, the thick client part that contains the business logic and the presentation logic must be re-written to run on a Web server. This will then mean that there are two copies of the business logic—one housed in the original thick client and the other housed in the Web-based version of the application. This is a nightmare in maintenance terms because any changes or updates must be made in both places. More decoupling is required to improve the manageability and maintainability of the application.

The decoupling of application logic by introducing additional tiers, as started with the two-tier system shown in Figure 1.2, can be continued with the separation of the Business and Presentation Logic. By housing the separated Business Logic in another tier, the thick client suddenly becomes thinner, as Figure 1.3 shows.

FIGURE 1.3

3-tier scenario.



The Presentation Logic is now separated into its own layer in its own tier. This means that different types of Presentation Logic, such as HTML-based and GUI-based user interface code, can all access the same Business Logic on the middle tier.

This 3-tier model has become the de-facto architecture for Web-based business systems. The separation into layers makes systems more flexible so that parts can be changed independently. An example of this would be creating a presentation layer specifically

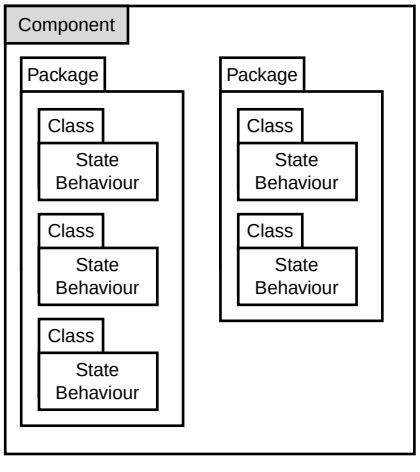
targeted at mobile devices. Given the separation of business and presentation functionality, this should not require any changes to the business of Data Access logic. The separation into separate physical tiers provides the opportunity to inject enhanced scalability and availability by replicating machines and software at the different tiers.

With the logic now separated into layers, it is far easier to write code that is tailored to its particular task. For example, because the Presentation logic is now housed in its own physical and logical layer, such code can be written by a developer who is skilled in this particular area. Developers who are skilled in the use of Java Web components, such as servlets (see Day 12, "Servlets") and Java Server Pages (JSPs) (see Day 13, "JavaServer Pages") can write the code for this layer. These developers do not need to know about the technologies used in the business or data access code.

Complexity Simplified by Modularity

When designing a system, certain concepts will naturally sit together. By placing these into shared modules, a certain amount of separation can be achieved that is independent of the layering discussed so far. Functionality can be split into classes, and these classes can be grouped in packages or components. By reducing the dependencies between the classes and packages, this functionality can be used by different parts of the application. By defining and maintaining interfaces between classes and packages, the actual implementation of a class can be replaced without requiring a change to other classes that depend on it. The Unified Modeling Language (UML) diagram in Figure 1.4 shows this type of decomposition.

FIGURE 1.4
Modularity.



Object-oriented (OO) modeling promotes modularity to a large extent. Objects encapsulate their data or state and offer functionality through their interfaces. If designed correctly, the dependencies between different objects can be minimized. This reduction in dependency means that the objects are loosely coupled. Loosely coupled systems tend to be easier to maintain and evolve.

Object-oriented programming tried to improve maintainability with encapsulation and to aid system design with a definition of specific classes for specific roles, providing coherent groups of functionality. This significantly improved the previously poorly designed monolithic code and made things more maintainable and flexible. However, it was language-specific (Java, C++, and Smalltalk) and so did not make deployment or integration easier.

Although it is not the whole solution, you have some useful tools for modularizing your applications in Java:

- A Java class is a way of adding modularity by housing all state and behavior belonging to an entity into one part of the design.
- A Java package is another way of using modularity to house all classes and interfaces that belong together to perform a specific set of functions.

What you then need is a way of going beyond simple objects to provide more coarse-grained packages of functionality that can be glued together to create custom applications. To be correctly glued, these packages must conform to certain rules that are defined by a framework. This leads us to components.

Component Technology

A *component* is a unit of functionality that can be used within a particular framework. Component frameworks have evolved to provide support for simplified application development. When using a component framework, a container provides the components with certain standard services, such as communication and persistence. Because standard mechanisms are used for component definition and inter-component communication, it becomes possible to write tools that examine components and display their information to an application writer. A developer can then use the tool to drag and drop these components into his or her application. This can be seen in the typical GUI interface builder environments, such as Visual Basic, or the Java equivalents, such as Borland's JBuilder and IBM's Visual Age for Java.

The component principle applies to non-visual components also. Whole distributed applications can be created from components. One of the benefits of distributed component frameworks is that they can provide language independence. Using CORBA, for example, components written in C can communicate with those written in OO languages such as Java and Smalltalk.

In Java, there are several component frameworks from which to choose. The J2EE platform uses components extensively to provide modularity within the layers of an application. As such

- A Java component is yet another way of using modularity to house all packages required to perform a specific task. In the 3-tier environment, for example, the functionality of the Data Access Logic layer would be split into multiple components.
- A component will publish its interface defining the functionality it offers. This functionality can then be used by the application itself or by other components.

Benefits of Modularity

If separate parts of the design can be identified, the most appropriate developers can be tasked with the implementation simultaneously. Some components can also be purchased from third parties and integrated quite easily because all components will conform to the framework. This brings down the time to market and is, therefore, a significant cost benefit.

The system is more maintainable if identifiable parts are capable of being upgraded and re-implemented without hindering the existing running of the system. With modularity comes the possibility of loose coupling, which means the system itself is extendable without introducing dependencies. If a module has loose coupling, its maintenance is simpler.

Using components within layers allows you to further modularize the functionality in those layers.

Benefits of the 3-Tier Scenario

A modern n-tier application architecture, such as that provided by J2EE, involves the separation of functionality both by using layers and tiers and also the use of components within those layers (and objects within those components).

Now, presentational developers need not know anything of the business rules in the system, and any changes to any of the layers should not impact the effectiveness of any of the others. This aids in maintenance of the system and promotes scalability and extensibility. The separation into components helps with the division of tasks even further.

With the advent of the Internet, enabling more businesses to deliver goods and services online, it is easier to deliver functionality to customers and business users. There may be issues with particular versions of browsers, but compared to the situation where thick

client applications would need to be distributed and installed on each client machine, the relative merit of distributing functionality using the Internet remains overwhelming considering the potential user base companies are striving to reach.

It is this emphasis on the Internet that many enterprise service vendors are seeking to exploit. Most organizations have some form of Web presence and many are trying to use this to offer services to their customers. Since such services interact directly with the customer, their levels of reliability and usability must be high. Such Web-based applications are now common currency, and the world is evolving further. Web services are being discussed as the next generation of n-tier development, allowing applications to be created from components distributed across the Internet. As this model evolves, the distributed Internet *becomes* the computer.

Enterprise applications can be Web centric, but need not be. To cover Web-centric programming, this book shows how to integrate Servlets (see Day 12, "Servlets") and Java Server Pages (JSPs) (see Day 13, "JavaServer Pages") into Enterprise applications. Within an organization, or even when creating business-to-business (B2B) links, enterprise applications need not use Servlets or JSPs. In this case, clients may connect directly to business components, in the shape of Enterprise JavaBeans (see Day 4, "Introduction to EJBs"), over RMI or CORBA.

As the provision of functionality over the Internet gains importance, most companies will expose the functionality of their internal applications as part of a Web-based application. As functionality is exposed, it becomes important to maintain the integrity of the data in the corporate systems. Transactions provide a common mechanism for doing this. Transactions are covered in detail in Day 8, "Transactions and Persistence."

A Model for Enterprise Computing

So, an n-tier, component-based, Web-friendly environment is needed, but what about the detail—what specific functionality is needed to support such applications? When considering what is needed for a distributed environment, we can turn to an organization that has been involved in this area for a long time. Since its inauguration in 1989, the OMG has been working with key industry players (such as 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems, and Unisys Corporation) to produce a component-based software market by hastening the introduction of standardized object software. Many of the OMG's specifications have become a standard part of the Distributed Object Computing landscape, such as the Common Object Request Broker Architecture (CORBA), the Internet Inter-Orb Protocol (IIOP), and the Unified Modeling Language (UML).

By examining some of the key requirements outlined in the OMG's Enterprise Computing Model, it is possible to explore what is required from a modern distributed computing environment.

Lifecycle

There must be a safe mechanism for creating, copying, moving, and deleting distributed objects. A distributed component environment must provide containers to manage the lifetime of components and assist in their deployment. There are also other lifecycle issues that must be addressed in a distributed environment. For example, distributed garbage collection (getting rid of unused objects) can be handled in different ways according to the operating environment. With Java's Remote Method Invocation (RMI), a distributed leasing mechanism is used. With CORBA, there are lifecycle management services. Microsoft's Distributed COM (DCOM), on the other hand, relies on objects controlling their own lifetimes.

Persistence

In an enterprise application, you need to be able to store data permanently for later retrieval. Object Database Management Systems (ODBMS) and Relational Database Management Systems (RDBMS) commonly support this requirement. A distributed application environment must provide a way of accessing and updating persistent data in a simple yet flexible way. It is also important to support different types of data persistence (different databases, legacy systems, and so on) and different ways of accessing this data (locally or across a network). Any help that can be given to the developer for data persistence is generally very welcome.

Naming

Distributed applications will be formed from components that reside on different machines. The parts of the application that use components on other machines must be able to locate and invoke such components. What is needed is a directory service in which components or services can register themselves. Any part of the application that wants to use such a service can look up the location of the service and retrieve information about how to contact it.

Common directory services and protocols include the following:

- *CORBA Common Object Services (COS) Naming Service*—This allows you to store object references in a namespace. The COS naming service is widely used in Java-based distributed environments as a way of storing information about the location of remote objects. Further information on COS Naming can be found online at `http://www.omg.org`.

- *X.500*—This defines an information model for storing hierarchical information and an access protocol called the Directory Access Protocol (DAP). Further information about X.500 can be found online at `http://java.sun.com/products/jndi/tutorial/ldap/models/x500.html`.
- *Lightweight Directory Access Protocol (LDAP)*—This is a lightweight version of the X.500 protocol that runs over TCP/IP. Further information on LDAP can be found online at `http://www.openldap.org`.
- *Domain Name System*—This is an Internet protocol that allows translation between host names and Internet addresses. DNS is used by all Internet clients, such as Web browsers. More information on DNS can be found at `http://www.dns.net/dnsrd/rfc/`.
- *Microsoft Active Directory*—The Active Directory service allows organizations to store central information on data and services within an enterprise. Further information on Active Directory can be found online at `http://www.microsoft.com/windows2000/technologies/directory/default.asp`.

Transaction

In a distributed enterprise application, certain business processes will involve multiple steps. For example, a typical exchange of goods or services for payment will need to take payment details, verify those payment details, allocate the goods to be shipped, arrange the shipping, and take the payment. At any stage, the customer might be interrupted or the server could crash, not completing the entire transaction. If that happens, the enterprise application must be able to retrieve the previous state to continue with the transaction at a later time or to roll back the transaction so that the system is restored to its original state.

Transaction services provide a way of grouping updates to data so that either all of the updates are performed or none of them are performed. A transaction coordinator will be responsible for ensuring this. Transaction information is persisted so that the state of a transaction can survive a system crash. Transactions can be propagated across distributed method calls and even across message-based systems.

Security

A secure enterprise application environment will provide the following:

- *Authentication*—Are you who you say you are?
- *Authorization*—Are you permitted to do things you are requesting to do?

In addition to this, many enterprise application environments will support both programmatic and declarative security. Programmatic security is enforced within the enterprise application itself, while declarative security is enforced by the enterprise application

environment within which the application runs. The enterprise application environment will consult configuration information to decide which security restrictions to enforce for a particular application. Changes to this information would not necessitate recompilation of the application itself.

Java 2 Enterprise Edition (J2EE)

J2EE is an on-going standard for producing secure, scalable, and highly-available enterprise applications. The standard defines which services should be provided by servers that support J2EE. These servers will provide J2EE containers in which J2EE components will run. The containers will provide a defined set of services to the components. The J2EE specification provides a definition from which enterprise vendors can produce J2EE application servers on which J2EE-compliant applications can be deployed. An impressive list of expert group members produced the latest version of the associated Java Specification Request (JSR 58 which contains the standard definition for J2EE version 1.3), which can be found online at `http://java.sun.com/j2ee/sdk 1.3/index.html`.

Although the J2EE specification defines a set of services and component types, it does not contain information on how to arrange the logical architecture into physical machines, environments, or address spaces.

The J2EE platform provides a common environment for building secure, scalable, and platform-independent enterprise applications. Many businesses are now delivering goods and services to customers via the Internet by using such J2EE-based servers. The requirements of such an environment demand open standards on which to build applications, for example,

- Java 2 Platform, Standard Edition (J2SE), a platform independent language
- Components that deliver Web-based user interfaces
- Components to encapsulate business processes
- Access to data in corporate data stores
- Connectivity to other data sources and legacy systems
- Support for XML, the language of B2B e-commerce

Components and Containers

J2EE specifies that a compliant J2EE application server must provide a defined set of containers to house J2EE components. Containers supply a runtime environment for the components. As such, Java 2 Platform, Standard Edition (J2SE) is available in each container. Application Programming Interfaces (APIs) in J2EE are also made available to

provide communication between components, persistence, service discovery, and so on. Containers are implemented by J2EE application server vendors and there should be a container available for each type of J2EE component:

- Applet Container
- Application Client Container
- Web Container
- EJB Container

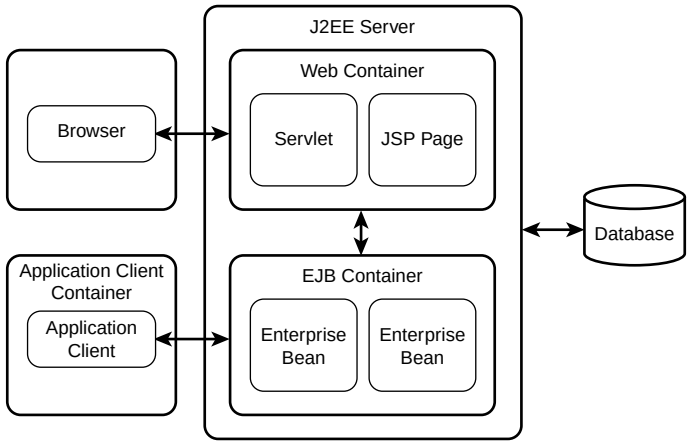
There are two types of components deployed, managed, and executed on a J2EE Server:

- *Web components*—A Web component interacts with a Web-based client, such as a Web browser. There are two kinds of Web components in J2EE—Servlet Component and Java Server Pages (JSP) Component. Both types handle the presentation of data to the user. Please see Days 12 and 13 for further details.
- *EJB components*—There are three kinds of Enterprise JavaBean components—Session beans, Entity beans, and Message-Driven Beans. Please see Day 4, Day 5, "Session EJBs," Day 6, "Entity EJBs," and Day 10, respectively, for further information.

Figure 1.5 shows the overall relationships between the different containers and components in the J2EE environment.

FIGURE 1.5

J2EE logical architecture.



J2EE Standard Services

Containers must provide each type of component with a defined set of services that are covered in detail as you progress through the book. Briefly these services consist of

- *Connectivity*—Containers must support connectivity to other components and to application clients. One form of required connectivity is to distributed objects through both Java Remote Method Invocation (RMI) and CORBA (as implemented by the Java IDL package and RMI over IIOP). Internet connectivity must be provided both through the Hypertext Transport Protocol (HTTP) and its secure form (HTTPS).
- *Directory services*—J2EE servers are required to provide naming services in which components can be registered and discovered. The Java Naming and Directory Interfaces (JNDI) provide a way of accessing these services.
- *Data access and persistence*—Data access is provided through the Java Database Connection API (JDBC). This API works both at the application level to interface with databases and also service providers who build drivers for specific databases.
- *Legacy connectivity*—The Java Connector Architecture (JCA or Connectors) provides J2EE support in integrating Enterprise Information Servers and legacy systems, such as mainframe transaction processing and Enterprise Resource Planning (ERP) systems. This support extends to J2EE service providers who are writing adapters to connect other systems to the J2EE enterprise architecture.
- *Security*—Security is built into the J2EE model. APIs, such as the Java Authentication and Authorization Service (JAAS), assist the J2EE enterprise application in imposing authentication and authorization security checks on users.
- *XML Support*—The JAXP API supports the parsing of XML documents using Document Object Model (DOM), SimpleAPI for XML documents (SAX), and the eXtensible Stylesheet Language Transformations (XSLT).
- *Transactions*—A J2EE server must provide transaction services for its components. The boundaries of transactions need to be specified by the container or the application. The container will usually take responsibility for transaction demarcation, although the Java Transaction API (JTA) allows the component to control its own transactions if required.
- *Messaging and e-mail*—The Java Message Service (JMS) allows components to send and receive asynchronous messages, typically within an organizational boundary. The JavaMail API enables Internet mail to be sent by components and also provides functionality to retrieve e-mail from mailstores. JavaMail uses the JavaBeans Activation Framework (JAF) to support various MIME types.

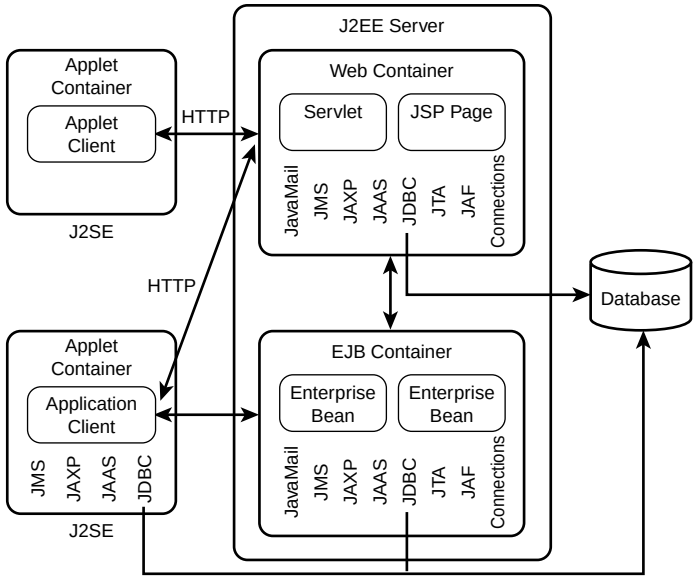
Figure 1.6 shows the J2EE architecture updated with the services available to its containers. All of these services are discussed in more detail tomorrow.

Every J2EE-compliant server must support the services defined in this section. To provide a concrete example of how services should work, the team working on the J2EE

JSR is responsible for providing a Reference Implementation (RI) of the J2EE APIs. This RI is freely available from Sun and provides a convenient platform for prototyping applications and testing technologies.

FIGURE 1.6

The J2EE platform with services available.



J2EE Blueprints

The J2EE Blueprints are a set of best practices that show how best to implement J2EE applications. The Blueprints provide a concrete implementation of Sun's vision for 3-tier, J2EE-based systems. There is a download available from the Sun Web site called Java Pet Store Sample Application that shows these best practices. This can be found online at `http://java.sun.com/j2ee/blueprints`.

The Java Pet Store is a typical online e-commerce application. The best practices cover application design and, in particular, the promotion of the following:

- Code reuse
- Logical functional partitioning
- The separation of areas of high maintenance
- Extensibility
- Modularity
- Security
- Simple and consistent user interface

- Efficient network usage
- Data integrity

The J2EE Blueprints will show you step-by-step how to design multi-tier enterprise applications. There are explorations on the following topic areas:

- The Client Tier
- The Web Tier
- The Enterprise JavaBeans Tier
- The Enterprise Information Systems Tier
- Design Patterns

In addition, there are discussions on how to package and deploy your enterprise applications.

J2EE Compatibility Test Suite

Enterprise service providers will sell more if their enterprise servers meet with the J2EE specification requirements. To enable them to test their products against the specification, Sun Microsystems Inc. offers a testing environment. Servers that pass all of the tests can be certified as J2EE compliant. Further details of the compatibility suite can be found online at `http://java.sun.com/j2ee/compatibility.html`.

The following are some examples of application vendors and their servers that have been certified by Sun Microsystems as being compatible with the J2EE specification (some of these may not yet have attained J2EE 1.3 certification, so please check on the Sun Web site).

- Allaire (`www.macromedia.com/software/coldfusion`)—ColdFusion 5 comes with its own markup language (ColdFusion Markup Language) that integrates with all popular Web languages and technologies. ColdFusion works with multi-tier architectures through COM, CORBA, and EJB integration.
- BEA Systems (`www.bea.com`)—BEA Weblogic Server includes support for Web Services, J2EE Connector Architecture, and updated J2EE services, with EJB 2.0, Servlet 2.3, and JSP 1.2.
- IBM (`www.ibm.com`)—Websphere Commerce Business Edition supports EJB, JSP, XML, HTTP, and wireless markup language technologies.
- iPlanet (`www.iplanet.com`)—iPlanet Application Server Enterprise Edition supports the J2EE platform and is integrated with transaction monitor, iPlanet Web Server, and iPlanet Directory Server. It supports XML, wireless application protocols, Simple Network Management Protocol (SNMP), LDAP, CORBA, and JDBC.

- JBoss (`www.jboss.org`)—JBoss is a freeware server that houses an implementation of the EJB 1.1 (and parts of 2.0) specification. It is similar to Sun's J2EE Reference Implementation, but the JBoss core server provides only an EJB server. JBoss does not include a Web container, but JBoss is available to download with a freeware Web server.
- Persistence (`www.persistence.com`)—PowerTier Release 7 for J2EE supports Java, EJB deployment, and Rational Rose integration.

The Future of J2EE

Probably *the* major area for the future of J2EE is that of Web Services. There are a number of JSRs active at the time of this writing on the following topic areas:

- *JSR 67 Java APIs for XML Messaging (JAXM) 1.0*—Message-based communication between Web Services and Web Service clients. Please refer to Day 21, "Web Service Registries and Message-style Web Services," for further details.
- *JSR 93 Java APIs for XML Registries 1.0 (JAXR)*—Registry and naming service access for Web Services. Please refer to Day 21 for further details.
- *JSR 101 Java APIs for XML-RPC (JAX-RPC)*—RPC-style interaction with Web Services. Please refer to Day 20, "Using RPC-Style Web Services with J2EE," for further details.
- *JSR 109 Implementing Enterprise Web Services*—A model of how Web Services should work within J2EE.

These JSRs can be found through the Java Community Process (JCP) Web site at `http://www.jcp.org`.

Summary

Enterprise application development has helped businesses provide Web-enabled, scalable, secure applications quickly. It has also enabled vendors to produce pluggable tools and services to augment the J2EE standard defined through the Java Community Process. This chapter describes the journey towards the n-tier environment that underpins the architecture of enterprise application programming. You have investigated the basic services that should be available to an n-tier enterprise application, and examined a few of the enterprise application servers on the market.

Q&A

Q I have a monolithic program that I would like transition into an n-tier application. How do I do this?

A First you need to identify what sort of target architecture is required. If your application is to be Web-enabled, you will need to provide Web-oriented functionality in the presentation layer. If you are working with persistent data, you will need data access through a data access layer. You should map out your target architecture based on the services available under the J2EE platform.

Next, you will need to sift through the monolithic code separating out the code belonging to the logical layers. This code might need to be rewritten in such a way as to make it maintainable and extensible. Introduce modularity by adopting object-oriented programming and design classes. Package these classes and design components to have maximum cohesion and loose coupling wherever possible.

To implement and deploy your J2EE application, read the rest of the book and follow the examples.

Q What is the difference between Microsoft's .NET framework and J2EE?

A You can build enterprise applications with both platforms. Both J2EE and .NET framework applications can provide good levels of scalability, availability and so forth. The essential difference is largely one of choice. J2EE lets you use any operating system, such as Windows, UNIX, or a mainframe. J2EE's development environment can be chosen to suit developers from a variety of Integrated Development Environment (IDE) and J2EE application server vendors. The .NET framework is essentially limited to the Windows family of operating systems. This allows it to be more cleanly integrated with the operating system, but reduces the choice of target platform.

Exercises

To extend your knowledge of n-tier development, try the following exercises:

1. Write a design for a monolithic application to provide a shopkeeper with data concerning stock information.
2. Redesign the application based on the contents of this chapter, so as to make it accessible over the Internet.
3. Visit `http://java.sun.com/j2ee` for further details of the J2EE programming tools and utilities.
4. Visit `http://www.microsoft.com` for further details of the .NET framework. Compare and contrast this with the facilities available under J2EE.

WEEK 1

DAY 2

The J2EE Platform and Roles

Yesterday, you learned about enterprise computing and some of the problems facing developers of enterprise solutions. The day also introduced J2EE, a technology that can help you develop secure, scalable, and platform-independent solutions that meet the needs of today's business.

Today, you will explore the J2EE platform and see what it can offer you to help solve your business problems. J2EE is a large framework that boasts of a wide-range of components, services, and roles. It is these that you explore today, so that you'll be eager and prepared to start writing code tomorrow. The following are the major topics today covers:

- Understanding how J2EE delivers solutions for today's business
- Introducing the available Web-centric components
- Introducing the use of Enterprise JavaBeans
- Assessing platform roles
- Exploring the packaging and deployment of enterprise applications

Revisiting the J2EE Platform

You learned a lot about enterprise computing yesterday. You learned specifically about how business needs force the evolution of application architectures; today, most applications are distributed across multiple machines. This approach, the n-tier model, gives rise to different ways of writing and structuring applications. Units of functionality—components—provide modularity that allow multiple developers to work more easily on different parts of the application. Use of a component framework also allows developers to apply third-party components to speed development. These loosely-coupled components may run as an application on a desktop client, within a Web server, or even on a server that connects to a legacy system. In addition, data has undergone a revolution. Data sources now go beyond simple, relational databases containing tables to encompass databases that contain serialized objects or plain text files containing XML. Alternatively, data may take the form of user information in an LDAP directory or information in an Enterprise Resource Planning (ERP) system.

Applications written in traditional programming languages that do not have supporting frameworks simply cannot perform the operations required by today's environment. Instead, you must employ component-aware programming languages together with frameworks dedicated to enterprise computing. As you have already seen, J2EE is such a framework. Although the environment within which J2EE operates might sound daunting, J2EE isn't. When you write J2EE applications, you still write Java code, and you still get to use the J2SE classes with which you are familiar.

To successfully use J2EE, you must

- Install and configure your J2EE environment
- Understand J2EE roles
- Appreciate the purpose of containers
- Understand how you can use J2EE components
- Understand the services that containers supply to components
- Learn or explore a new set of APIs

Yesterday's lesson introduced the first four points in the list. You will explore them in more depth today. After you understand these, you will be ready for tomorrow, when you will start to apply the new APIs and to code real applications against them.

Using Sun Microsystems' J2EE SDK

Before you can start coding real J2EE applications, you need a J2EE implementation and a Java development environment, such as Sun Microsystems JDK or a Java Integrated

Development Environment (IDE). This book uses the Sun Microsystems' J2EE SDK, which is a complete reference implementation of J2EE. It includes all the classes, containers, and tools you need to learn J2EE.

To run the example code provided on the CD-ROM accompanying this book, you will also need to install a sample database. Installing the sample database is described in the Exercise at the end of this day's lesson. But now, to give you some hands-on work before you study the theory behind J2EE, you will install the J2EE SDK 1.3 on your workstation.

The J2EE SDK is free to download, use for learning J2EE, and use as a development tool. The Sun Microsystems license for this product states that you may not use it in a production environment. Be warned!

Installing J2EE SDK 1.3

Before you download the SDK, ensure that you have J2SE 1.3.1_01 (also known as JDK 1.3.1) or later correctly installed and are using one of the following supported platforms:

- Windows NT4 or 2000
- Solaris SPARC 7 or 8
- Linux Redhat v 6.0 or 6.1

You can use a Java IDE that supports J2SE 1.3 (or later) in preference to the Sun Microsystem's J2SE JDK 1.3.1.

Before installing J2EE SDK 1.3, you must uninstall any previous versions of the J2EE SDK.

Finally, you must ensure that you have a `JAVA_HOME` environment variable that points to the location of the directory where you installed J2SE SDK (or your preferred Java IDE). This should have been defined when you (or your administrator) installed the J2SE SDK (JDK). If the `JAVA_HOME` variable is not defined, you must define it now as follows.

If you are using Windows NT or 2000 (remember J2EE SDK is not supported on other Windows' platforms) you should set the `JAVA_HOME` variable in your system environment so that it is defined for all of the programs you run. Do this using the Control Panel as follows:

1. Within the Control Panel, select System.
2. The System Properties dialog appears, select the Advanced tab.
3. Click Environment Variables.

4. The Environment Variables dialog appears, click New.
5. The New System Variable dialog appears, enter the name and value of the variable. Assuming that you installed the J2EE SDK 1.3 on the C: drive using the default directory name, you will set the `JAVA_HOME` variable to

   ```
   C:\j2sdkee1.3
   ```

6. Click OK to clear each of the dialogs.

   You must have administrator privileges to edit or change system environment variables. If you do not have administrator privilege, you can still use the JDK, but you'll have to define the variables in your user environment. Any other users of your workstation will also have to define the same variables in their environment.

If you are using Linux or Unix and the `JAVA_HOME` environment variable does not exist, you can set it with the following command(you must be using the Bourne, Korn, Bash or compatible shell):

```
JAVA_HOME=/usr/local/jdk1.3.1
export JAVA_HOME
```

This example assumes you have installed the Sun Microsystems' JDK 1.3.1 in `/usr/local`.

Typically, you will add these variable definitions to your login environment by adding the same two lines to the `.profile` file in your home directory.

Finally, you should ensure that the JDK `bin` directory is in your search path (again this should already be configured on your workstation).

For Windows users, if your search path does not contain the JDK `bin` directory, add the following directory to your `PATH` via the Control Panel:

```
%JAVA_HOME%\bin
```

For Linux/Unix users, if your search path does not contain the JDK `bin` directory, add the following line to your `.profile` file:

```
PATH=$PATH:$JAVA_HOME/bin
```

Now download the J2EE SDK in the format appropriate to your system from `http://java.sun.com/j2ee/sdk_1.3/index.html`. You should download the J2EE SDK to a temporary directory because the installation process will ask you where to install the SDK.

The installation of the SDK is quite straightforward; just follow the instructions for your platform:

- *Windows*—Double-click the icon of the `j2sdkee-1_3_01-win.exe` file and follow the onscreen instructions.
- *Solaris*—Issue the following commands to make the download bundle executable and run the installation::

```
chmod a+x ./j2sdkee-1_3__01-solsparc.sh
./j2sdkee-1_3__01-solsparc.sh
```

- *Linux*—Change directories to the required parent directory for the J2EE SDK (for example, `/usr/local`) and extract the download bundle using the following command:

```
tar – xzvf <download_directory>/j2sdkee-1_3_01-linux.tar.gz
```


Now you must:

1. Define the `J2EE_HOME` variable.
2. Add the J2EE SDK `bin` directory to your search path.
3. Add the J2EE classes to your `CLASSPATH`.

You have already been shown how to define variables and change your path for your environment (Windows users use the Control Panel and Linux/Unix users add variable definition lines to `.profile`), so making these changes will be straightforward.

Windows users must (assuming the J2EE SDK was installed on the `C:` drive)

1. Define the following environment variable

   ```
   J2EE_HOME=c:\j2eesdk1.3
   ```

2. Add the following directory to the end of the `PATH` variable:

   ```
   %J2EE_HOME%\bin
   ```

3. Add the J2EE JAR files to the `CLASSPATH` variable:

   ```
   %J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale
   ```

   If your `CLASSPATH` variable is not currently defined, you must ensure that it includes the current directory so the full setting will be

   ```
   .;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale
   ```

Linux/Unix users must add the following to their `.profile` (assuming the J2EE SDK was installed in `/usr/local/j2eesdk1.3`):

1. Define the following environment variable:

   ```
   J2EE_HOME=/usr/local/j2eesdk1.3
   ```

2. Update the `PATH` variable for J2EE SDK:

   ```
   PATH=$PATH:$J2EE_HOME/bin
   ```

3. Add the J2EE JAR files to the `CLASSPATH` variable:

   ```
   CLASSPATH=$CLASSPATH:$J2EE_HOME/lib/j2ee.jar:$J2EE_HOME/lib/locale
   ```

   If your `CLASSPATH` variable is not currently defined, you must ensure that it includes the current directory so the full setting will be:

   ```
   CLASSPATH=.:$J2EE_HOME/lib/j2ee.jar:$J2EE_HOME/lib/locale
   ```

That is it! You are now ready to start using the J2EE SDK.

All the documentation for the J2EE utility programs and class files is contained in the J2EE SDK download bundle. You will find the documentation in the `docs` sub-directory of the J2EE installation directory.

Starting the J2EE Reference Implementation (RI)

The J2EE SDK includes a Reference Implementation of J2EE. This Reference Implementation (RI) contains the following software programs:

- A J2EE server
- A relational database called Cloudscape
- An HTTP (Web) server
- A JNDI service implementation
- A JMS service implementation
- Class files for the J2EE APIs
- Various administration and support utilities

The server software components of the RI (database, JNDI, Web server, and J2EE server) are purely for development and are not designed for commercial use. The J2EE RI has been used for the code shown in this book because it is free of charge and conforms to the J2EE 1.3 specification.

To develop J2EE applications and run the example code presented in this book, you will need to start the J2EE server and the Cloudscape database server. Starting the J2EE server will also start the JNDI, JMS, and HTTP servers provided with the J2EE RI.

There are no graphic tools for managing the J2EE and Cloudscape servers, so you will have to start them from the command line. Each server should be started in a separate command window or Telnet window if you are not using a graphical console. It does not matter in which order you start the J2EE and Cloudscape servers.

In the following examples, you must have configured your search path (the `PATH` environment variable) to include the J2EE SDK `bin` directory as shown previously.

To start the J2EE RI server, create a new window with access to a command-line prompt (command window for Windows NT/2000 or a terminal or shell window for Linux/Unix users). Enter the following command at the prompt:

```
j2ee –verbose
```


This will start the J2EE server in the current window with diagnostic messages displayed in the window. If (or when) you have problems deploying your J2EE applications to the server, it is this window you should examine for error messages. If you put simple diagnostic messages in your EJBs that write to `System.out` or `System.err`, these messages will also appear in this window.

Listing 2.1 shows the diagnostic messages issued as the J2EE RI and associated servers startup.

LISTING 2.1 Successful J2EE Reference Implementation Startup Diagnostics

```
   1: > j2ee -verbose
   2: J2EE server listen port: 1050
   3: Naming service started:1050
  4: Binding DataSource, name = jdbc/DB2,
➥url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
   5: Binding DataSource, name = jdbc/DB1,
➥url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
   6: Binding DataSource, name = jdbc/InventoryDB,
➥url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
   7: Binding DataSource, name = jdbc/Cloudscape,
➥url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
   8: Binding DataSource, name = jdbc/EstoreDB,
➥url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
   9: Binding DataSource, name = jdbc/XACloudscape, url = jdbc/XACloudscape__xa
  10: Binding DataSource, name = jdbc/XACloudscape__xa, dataSource = COM.cloud-
scape.core.RemoteXaDataSource@653220
  11: Starting JMS service...
  12: Initialization complete - waiting for client requests
  13: Binding: < JMS Destination : jms/Queue , javax.jms.Queue >
  14: Binding: < JMS Destination : jms/firstQueue , javax.jms.Queue >
  15: Binding: < JMS Destination : jms/Topic , javax.jms.Topic >
  16: Binding: < JMS Cnx Factory :
➥QueueConnectionFactory , Queue , No properties >
  17: Binding: < JMS Cnx Factory :
➥jms/QueueConnectionFactory , Queue , No properties >
  18: Binding: < JMS Cnx Factory :
➥TopicConnectionFactory , Topic , No properties >
  19: Binding: < JMS Cnx Factory :
```

LISTING 2.1 Continued

```
➥jms/TopicConnectionFactory , Topic , No properties >
  20: Starting web service at port: 8000
  21: Starting secure web service at port: 7000
  22: J2EE SDK/1.3
  23: Starting web service at port: 9191
  24: J2EE SDK/1.3
  25: J2EE server startup complete.
```

If you start the J2EE server without the `-verbose` option, all the diagnostic messages will be written to log files in the `logs` sub-directory of the J2EE SDK installation directory. You will find additional logging information is also written to these log files even with the `-verbose` option specified.

The J2EE log files are stored in a sub-directory named after your workstation in the logs sub-directory of the J2EE SDK installation directory. Further sub-directories are used to separate the log files for the J2EE, JMS, and HTTP servers.

To start up the Cloudscape database server, you must open a new window and enter the following command:

```
cloudscape –start
```

Again, some simple diagnostic messages will be displayed as the server starts up, as shown in Listing 2.2.

LISTING 2.2 Successful Cloudscape Startup Diagnostics

```
> cloudscape -start
Thu Jan 10 10:52:18 GMT+00:00 2002:
➥ [RmiJdbc] Starting Cloudscape RmiJdbc Server
 Version 1.7.2 ...
Thu Jan 10 10:52:25 GMT+00:00 2002:
➥ [RmiJdbc] COM.cloudscape.core.JDBCDriver registered in DriverManager
Thu Jan 10 10:52:25 GMT+00:00 2002: [RmiJdbc] Binding RmiJdbcServer
Thu Jan 10 10:52:25 GMT+00:00 2002:
➥ [RmiJdbc] No installation of RMI Security Manager
Thu Jan 10 10:52:26 GMT+00:00 2002:
➥ [RmiJdbc] RmiJdbcServer bound in rmi registry
```

Troubleshooting J2EE and Cloudscape

You should have no problems starting up either server. If you do have problems, check the error messages displayed in the relevant window. The most likely problems are discussed in the rest of this section.

Read Only Installation Directory

You will not be able to run J2EE RI and Cloudscape unless you have installed the J2EE SDK in a writeable directory. If you have installed the J2EE SDK as a privileged user (Administrator, root, or whomever), make sure that you grant your normal login account read and write permission to the installation directory and all contained files and directories.

Server Port Conflicts

Although the J2EE SDK software uses TCP port numbers that are not normally used by other software, there is always a possibility that there will be a port number conflict.

If a port is used by another software server, a J2EE component will fail to start up and you will see an error message stating that the server "Could not connect to a required port." The error will normally include a stack trace.

The most likely cause of a port conflict is where you (or another developer) have already started the J2EE server. Listing 2.3 shows the error message for this situation.

LISTING 2.3 Error Message Caused by Running the J2EE Server Twice

```
> j2ee -verbose
J2EE server listen port: 1050
org.omg.CORBA.INTERNAL:   minor code: 1398079697  completed: No
at com.sun.corba.ee.internal.iiop.
➥GIOPImpl.createListener(GIOPImpl.java:256)
        at com.sun.corba.ee.internal.iiop.
➥GIOPImpl.getEndpoint(GIOPImpl.java:205)
        at com.sun.corba.ee.internal.iiop.
➥GIOPImpl.initEndpoints(GIOPImpl.java:140)
        at com.sun.corba.ee.internal.POA.POAORB.
➥getServerEndpoint(POAORB.java:488)
        at com.sun.corba.ee.internal.POA.POAImpl.
➥pre_initialize(POAImpl.java:154)
        at com.sun.corba.ee.internal.POA.POAImpl.<init>(POAImpl.java:112)
        at com.sun.corba.ee.internal.POA.POAORB.makeRootPOA(POAORB.java:110)
        at com.sun.corba.ee.internal.POA.POAORB$1.evaluate(POAORB.java:128)
        at com.sun.corba.ee.internal.core.Future.evaluate(Future.java:21)
at com.sun.corba.ee.internal.corba.ORB.
➥resolveInitialReference(ORB.java:2421)
        at com.sun.corba.ee.internal.corba.ORB.
➥resolve_initial_references(ORB.java:2356)
        at com.sun.enterprise.server.J2EEServer.run(J2EEServer.java:193)
        at com.sun.enterprise.server.J2EEServer.main(J2EEServer.java:913)

java.lang.RuntimeException: Unable to create ORB.
➥ Possible causes include TCP/IP ports in use by another process
```

LISTING 2.3 Continued

```
        at com.sun.enterprise.server.J2EEServer.run(J2EEServer.java:203)
        at com.sun.enterprise.server.J2EEServer.main(J2EEServer.java:913)
java.lang.RuntimeException: Unable to create ORB.
➥Possible causes include TCP/IP ports in use by another process
        at com.sun.enterprise.server.J2EEServer.run(J2EEServer.java:203)
        at com.sun.enterprise.server.J2EEServer.main(J2EEServer.java:913)

java.lang.RuntimeException: Unable to create ORB.
➥Possible causes include TCP/IP ports in use by another process
        at com.sun.enterprise.server.J2EEServer.run(J2EEServer.java:350)
        at com.sun.enterprise.server.J2EEServer.main(J2EEServer.java:913)
J2EE server reported the following error: Unable to create ORB.
➥Possible causes include TCP/IP ports in use by another process
Error executing J2EE server ...
```

You cannot run more than one J2EE RI server on the same workstation.

If your port conflict is with another piece of software, try to disable this software when running J2EE RI and Cloudscape. If this is not possible, you can change the port numbers used by each J2EE server by editing the file called `server.xml` in the `conf` subdirectory of the J2EE SDK installation directory. The definitions of the default server port numbers are obvious if you are used to reading and editing XML files. Changing the J2EE RI port numbers is something you should avoid if at all possible.

Applications Failing with a "`Connection Refused: no further information`" Exception

A common error when working with the J2EE RI is to forget to start up the Cloudscape database server in a separate window. If you fail to start the database and use a J2EE component, such as an EJB, that accesses the database, you will get the following error:

```
java.sql.SQLException: Connection refused to host:
➥<hostname>; nested exception is:
java.net.ConnectException: Connection Refused: no further information
```

To solve this problem, start up the Cloudscape server as described previously and stick a note to your monitor to remind you to start the database as well as J2EE. If you are confident with batch or shell scripts, you can write your own scripts to start both servers in separate windows.

The following simple scripts will work for the indicated platforms:

On Windows NT/2000, use

```
start "J2EE" j2ee -verbose
start "Cloudscape" cloudscape -start
```

On Solaris, use

```
dtterm -name j2ee -e "j2ee -verbose" &
dtterm -name cloudscape -e "cloudscape -start" &
```

On Linux, use

```
xterm -title j2ee -e "j2ee -verbose" &
xterm -title cloudscape -e "cloudscape -start" &
```


Closing Down J2EE RI and Cloudscape

To close down J2EE RI and Cloudscape, you should use the following commands:

```
j2ee -stop
cloudscape -stop
```

These commands can be run from any command window. Remember that the J2EE and Cloudscape server windows are busy and cannot accept additional commands while the servers are running.

Although Sun Microsystems do not recommend this approach, typing ^C (Ctrl+C) in the server window or simply closing the window will also shut down the servers.

Optional Software Used in this Book

A Java IDE (JDK) and a J2EE implementation (J2EE SDK) are all you require to learn how to develop J2EE applications. However, areas of this book look at using J2EE applications in a wider context and make use of additional (freely available) software.

So that you are aware of this software, Table 2.1 lists the optional software used in this book. Full instructions for downloading and configuring this software (should you want to do so) are included in the relevant day's instructions. You do not need to download this software at the present time.

TABLE 2.1 Optional Software Used in Daily Lessons

Day	*Software*	*Resource URL*
3	OpenLDAP and a Unix system to run the Open LDAP server.	`http://www.openldap.org/software/download/`
14	JSPTL Java Standard Tag libraries from the Apache Jakarta project.	`http://jakarta.apache.org/taglibs/index.html`
17	XALAN from the Apache project.	`http://xml.apache.org/xalan-j/index.html`

TABLE 2.1 Continued

Day	*Software*	*Resource URL*
20	Apache Axis alpha2.	`http://xml.apache.org/axis/index.html`
	Apache Tomcat 4.0.1.	`http://jakarta.apache.org/tomcat/index.html`
	JAXM 1.0 reference implementation (part of the "JAX Pack Fall 01").	`http://java.sun.com/xml`

Understanding Tiers and Components

The 3-tier model splits an application down so that business logic resides in the middle of the 3 tiers. This is often called the middle tier, business tier, or EJB tier. Throughout this section, this tier will be referred to as the business tier. The first tier has the role of providing the interface between the user and the application. Depending on your specific architecture, this can be known as the client tier or presentation tier. Throughout this section, the term presentation tier will be used to define this client-centric tier.

Most of the code written by J2EE application developers resides in the presentation and business tiers. The next two sections explore these tiers. You will see that different types of components are used in each tier to deliver particular application functionality. The components in different tiers should be loosely coupled. In other words, a component should not have dependencies on the client or other components. For example, imagine a business component that processes credit card payments. If the component is self contained, almost any application can pass it payment information and it, in turn, can return an appropriate response. Figure 2.1 shows such a business component communicating with a variety of clients:

As you can see, because this business component encapsulates all the payment process functionality, it is not tied to any component in the presentation tier, so it can serve multiple client types. Code that supports more than one type of client is only one advantage of component architecture; you will learn about some of the other advantages later in this chapter.

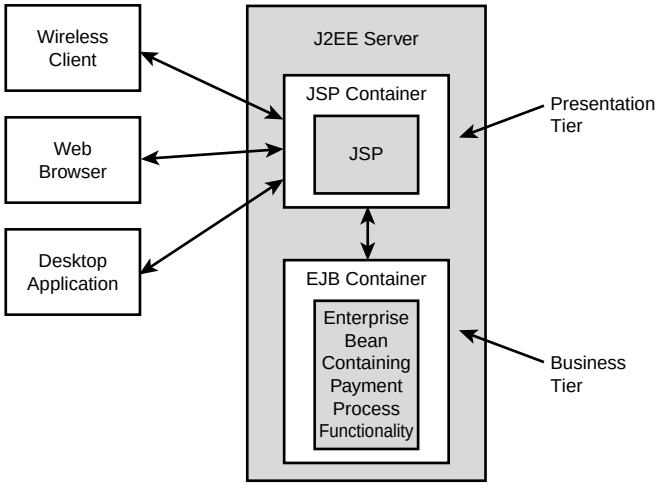


FIGURE 2.1
Multiple clients accessing the services of a business component.

The Business Tier

As you have just seen, business components sit in the business tier of a J2EE application. These business components encapsulate business logic and are used by components in the presentation tier that deliver this functionality to users of the application.

Benefits of Business Components

Previously, you saw a simple credit card processing component that provided that service for a number of different clients. This demonstrated just one benefit of component architecture but, in fact, components offer many advantages over a composite architecture:

- *Increased efficiency*—The division of labor helps a business to roll out applications quickly. The use of components allows presentation developers to develop GUIs, process programmers to focus on business logic, and data access experts to focus on data access.
- *Extensibility*—You can simply add or remove components to an application so that it can offer further functionality, for example, if you want to expose application functionality through a Web Service. The component architecture allows you to simply add the additional units of functionality the system requires. By the way, if you don't know about Web Services, don't worry. Day 20, "Using RPC-Style Web Services with J2EE," and Day 21, "Web Service Registries and Message-Style Web Services," show you how to apply a J2EE application as a Web Service.
- *Language independence*—A modularized system allows you to write code in one language that communicates with code written in another. For example, you can

access the functionality of a J2EE application from a CORBA client by using RMI-IIOP, or from a Microsoft COM client by using the Client Access Services COM Bridge.

- *System upgrade*—Inevitably, an organization's business processes change, and so too must the application logic. The use of components allows you to change one component without affecting the other components in the system.

This list is not an exhaustive survey of the benefits of components. But it should make it clear that components offer both developers and business an ideal way of providing application functionality.

J2EE defines how various types of components perform specific roles in different tiers. In the presentation tier, different types of components will be applied to provide functionality for different types of client (application components, applet components, servlet components, and JavaServer Page components). The business tier houses different types of business components. In J2EE terms, business components are embodied as Enterprise JavaBeans. The following sections focus on how such components are applied for the most common application architecture currently—namely, a business system with a Web-based user interface.

Components: Enterprise JavaBeans

In a typical business application built on the J2EE platform, business logic will be encapsulated in Enterprise JavaBeans (EJBs). It is possible to use other types of component or plain Java objects to implement business logic, but EJBs provide a convenient way of encapsulating and sharing common business logic, as well as benefiting from the services provided by the EJB container.

Suppose you were tasked with designing and implementing a typical Web-based, e-commerce application. Although the precise analysis of the business problem would be specific to your own environment, you would probably end up with the following flow through your application:

1. Display your products to the customer.
2. Allow the customer to select one or more products.
3. Confirm the order and take shipping details.
4. Take payment for the items.
5. Deliver the order to your warehousing and distribution systems.
6. [optional] Authenticate the user to access previously stored personal information or preferences.

7. [optional] Generate a report from the items purchased by a particular customer or on a particular day.

All of these steps involve a certain amount of business logic and data manipulation. For example, in step 1, your application will need to send pages of HTML to the customer's browser containing product descriptions and pricing information. To do this, you will need to retrieve this product and pricing information from somewhere. The obvious choice is a database that stores your product catalog information together with pricing information.

As you will see, it would be quite possible to deliver the catalog functionality you require simply by using database access code from a Web component such as a JavaServer Page. However, what if gathering this catalog information was not quite so straightforward?

- The product and pricing information may be spread across multiple databases. Even worse, it may be that some of the information must be extracted from (or delivered to, in the case of submitting an order) a legacy system. This means that the user-interface component would have to know details about data access.
- There may be extra business processes that need to be applied during the creation of the catalog. These could range from custom pricing for a specific group or individual through cross-selling of related products and on to the suggested substitution of alternative products for any that are not in stock. This means that the user interface component must have knowledge of multiple business processes.
- If a customer is to be identified and their preferences reflected, authentication is required. Also, some information about them and their preferences must be maintained against a database somewhere. This means that the user interface component must know about authentication and mapping user identity to stored information.

As you can see, the user interface component rapidly becomes its own version of the monolithic applications you saw yesterday. The business logic associated with all of this processing should be devolved to EJBs in order that the user interface component can concentrate on what it does best, namely generating a compelling user interface and guiding the user through a particular interaction.

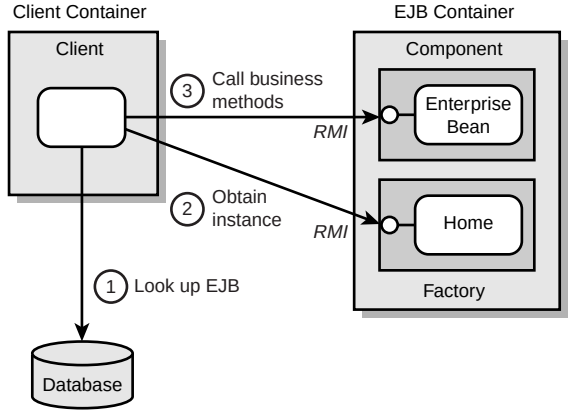
What is needed then is for the business logic in the EJB to be made available to other components that may want to use it. The mechanisms involved must be commonly available to the potential clients to impose minimal overhead on those clients. Hence, the EJB model makes use of two mechanisms found in J2SE—namely Remote Method Invocation (RMI) and the Java Naming and Directory Interface (JNDI)—to facilitate interaction between the EJB and its client. When an EJB is written, the functionality it

offers to clients is defined as an RMI remote interface. When an EJB is deployed, its location is registered in the naming service.

A client will then use JNDI to look up the location of the EJB. It will interact with a factory object, called the EJB's home, to obtain an instance of the EJB. This is equivalent of using `new` to create an instance of a local object. When the client has a reference to the EJB, it can use the business functionality offered by the EJB. This sequence is shown in Figure 2.2.

FIGURE 2.2

A client uses JNDI and RMI to access an EJB.

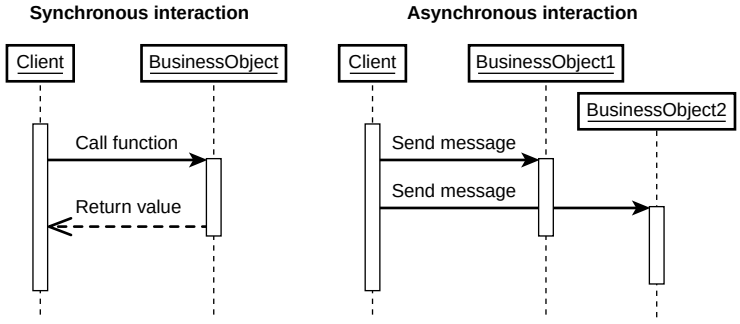


There is a more detailed look at how to use EJBs on Day 4, "Introduction to EJBs."

Within the required business logic, certain components will be primarily concerned with data and the manipulation of that data, whereas others will focus on the sequencing of business logic and the associated workflow. Equally, components will interact in different ways, depending on the needs of the application. Some interactions will be synchronous in nature; there is no point in performing the next stage of the process until the current one has finished. Other interactions can be handled asynchronously; an appropriate message can be sent to another component and then the originator of the message can carry on to the next stage of the process. This means that clients using asynchronous interactions complete faster, but they may not be suitable for all applications if there must be a guarantee that an operation has completed or if return values are required. This difference between synchronous and asynchronous interactions is shown in Figure 2.3.

FIGURE 2.3

Synchronous interactions will result in the caller waiting for the function to complete, whereas asynchronous interactions allow callers to proceed without waiting.



Because different business components are called on to behave in different ways, there are multiple types of EJB defined that can be used to encapsulate different parts of an application's business logic.

Components: Session Beans

Session EJBs, often just called *session beans*, are the simplest and probably most common type of EJB. A session bean is primarily intended to encapsulate a set of common business functions. When capturing the outputs of business analysis using the Unified Modeling Language (UML), Use Cases are identified. A *Use Case* documents a particular interaction sequence between a user and a system (a common example is withdrawing money from an ATM). Such an interaction typically involves multiple, but related, steps. The business logic associated with the steps from such a Use Case can typically be housed in a session bean. There are various analogies for and examples of session beans on Day 4 and Day 5, "Session EJBs."

Session beans offer a synchronous interface through which the client can use the business logic. Session beans are not intended to hold essential business data. The session bean will either hold no data on an ongoing basis, or the data it does hold will tend to be temporary (only relevant to the current user session) rather than persistent. If a session bean wants to obtain data from a database, it can use JDBC calls. Such a bean may provide part of solution when providing an e-commerce catalog as described earlier. Alternatively, application data can be obtained through entity EJBs.

Components: Entity Beans

An entity EJB, again often just called an *entity bean*, is a representation of some business data. During analysis, various business concepts will be discovered, such as "customer" or "account." These business "objects," sometimes called "entities," represent the core data manipulated during the business processes. If such a business object contains

dynamic data, has associated functionality, and can be shared between multiple clients at any one time, this business object would probably map to an entity bean.

Entity beans offer a synchronous interface through which the client can access its data and functionality. Entity beans will access underlying data sources (frequently, a database or possibly an ERP system) to collect all the business information they represent. The entity bean will then act as the dynamic representation of this business data—providing methods to update and retrieve it in various ways. As you will see later, entity beans are frequently used together with session beans to provide the business functionality of a system. Entity beans are discussed further on Day 6, "Entity EJBs."

A message-driven EJB, or just *message-driven bean*, fulfills a similar purpose to a session bean, but is asynchronous in nature. There are times when it is inefficient to interact synchronously with a component. One example would be if you wanted to log the details of a particular transaction to an underlying data store. In many cases, it is not important that the logging is done immediately, just as long as it is done reliably. This type of operation can quickly become a performance bottleneck if performed synchronously. The same is true of "undoable" operations, such as credit card processing.

Components: Message Beans

Message-driven beans offer an asynchronous interface through which clients can interact with them. The bean is associated with a particular message queue, and any messages arriving on that queue will be delivered to an instance of the message-driven bean. As with session beans, message-driven beans are intended to house business logic rather than data, so they will access any data required through JDBC or entity beans. To use the services of a message-driven bean, a client will send a message to its associated message queue. If a response is required, another message queue will typically be used. Message-driven beans and the Java Message Service with which they interact are discussed further on Day 9, "Java Message Service," and Day 10, "Message-Driven Beans."

The Presentation Tier

Given some business logic implemented as EJBs, you must provide clients with access to this functionality. What is needed is some presentation logic coupled with a way of displaying information to the user. The presentation logic will govern which screens are displayed to the user in which order and how the presentation logic will interact with the business logic in the business tier to work through the appropriate business process.

The way in which user input is received and information is displayed will depend on the type of client. The client can range from a WAP phone through to a standalone Java application with Swing interface. Each type of client will require different

presentation tier functionality to exchange and display information. The most common type of client for a J2EE application is a Web client, in the shape of a Web browser. In this case, the presentation tier will present a Web-based interface that produces HTML and consumes HTML form-based input.

Components: Web-Centric

To create a Web-based user interface, you need to apply Web-centric components. J2EE provides two types of Web-centric components—JavaServer Pages (JSP) and Java servlets. These components are applied in the presentation tier and provide services to clients that use HTTP as a means of communication. For example, Web-centric components can interact with clients such as the following:

- Standard HTML-oriented browsers, such as Microsoft Internet Explorer and Netscape Navigator
- Java 2 Micro Edition (J2ME) enabled devices, connecting across a wireless network
- Desktop clients using raw HTTP or sockets functionality
- Wireless Markup Language (WML) browsers, such as those found on WAP-enabled mobile phones

Figure 2.4 shows how you would typically use these components.

FIGURE 2.4

Typical use of Web-centric.components.

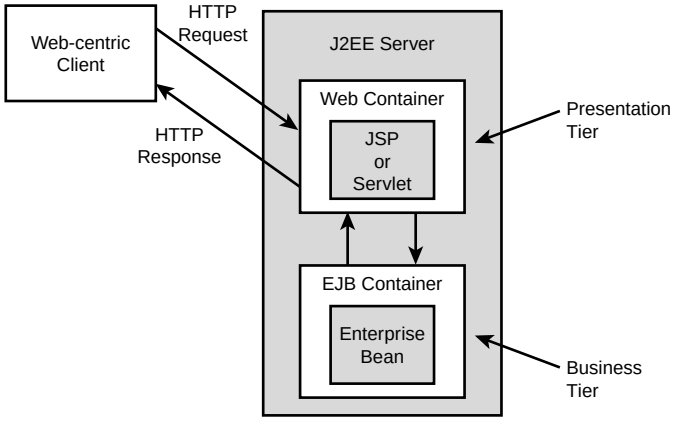


Figure 2.4 shows a Web-centric client making a request to either a servlet or JSP. The server-side component parses the client's request and then calls the EJB. The EJB contains the application's business logic. When the servlet or JSP receives a response from the EJB, it is responsible for presenting the data it receives. After the servlet or JSP has

prepared the response, it passes it back to the client, completing the request-response cycle.

The previous illustration demonstrated a model where the Web-centric component was responsible for presenting data supplied by an EJB. The EJB was responsible for the execution of the business logic. However, you do not have to use EJBs as part of a Web-centric solution. A simpler application can consist of pages of markup—for example, HTML—and servlets, or JSPs, or a combination of JSPs and servlets.

The next section of today's lesson shows you the relationship between JSPs and servlets, so that you are in a position to choose which of these technologies best suit your needs.

JavaServer Pages (JSP)

JSPs allow you to dynamically create pages of markup, such as HTML, in response to a client's request. If you are familiar with other Internet technologies, you can liken JSP to Active Server Pages (ASP) or ColdFusion. But be aware that these technologies are similar to JSP—not the same.

A JSP consists of a combination of JSP tags and scriplets, which contain the executable code, and static markup, such as HTML or XML. The code contained in the JSP is identified and executed by the server, and the resulting page is delivered to the client. This means that the embedded code can generate additional markup dynamically that is delivered to the client alongside the original static markup. The client sees none of this processing, just the result.

The JSP tags delimit sections of executable code and form the basis of any JSP page. Scriplets are delimited sections of script that allow a JSP further processing power. Typically, you can write this script using the Java programming language, but different JSP implementations may support additional languages.

Day 13, "JavaServer Pages," shows you in detail how to write JSPs and how JSPs interact with other J2EE components. To whet your appetite today will introduce you to how JSPs work and interact with other J2EE components. Figure 2.5 shows a scenario that depicts the typical interactions of client, JSP, and J2EE components.

Figure 2.5 shows a client making a HTTP request for a JSP. The first time a user makes a request the JSP container handles it by converting the JSP into a Java source file and compiling it. In most implementations, this file is a servlet. The servlet forwards the request to a business logic component, such as another servlet, a JavaBean, or an Enterprise JavaBean. The component performs some action, such as accesses a database or processes the client's input, and returns a response to the servlet. The servlet passes this response to a JSP that generates the markup language that the client will display. Finally, the JSP container and the Web Server return the markup to the client.

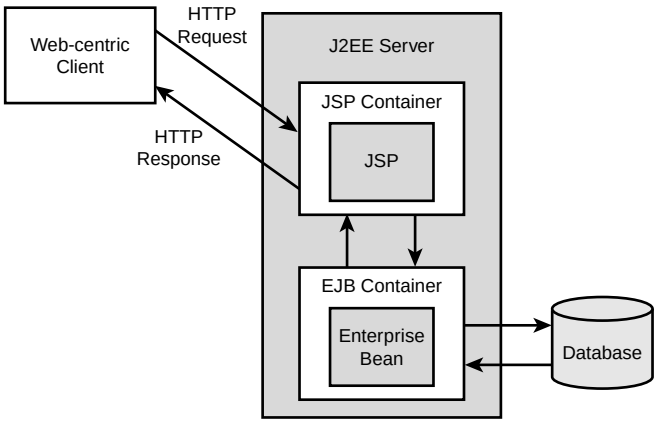


FIGURE 2.5
The interactions between a client, JSP, and J2EE component.

As you can see, JSPs provide a very powerful method for dynamically creating pages of markup. They also allow Web clients to indirectly access the application logic that other J2EE components contain. Importantly, you have seen the relationship between a JSP and a servlet.

Java Servlets

Servlets add processing power to servers that employ a response-request model. Perhaps the most common of such servers is the Web server. In this instance in the past, CGI scripts would provide this kind of functionality; now you can use servlets. Although servlets are similar to CGI scripts, they are superior in many ways:

- *Speed*—You deploy servlets as Java class files. The class file consists of Java byte codes, which means that they execute faster than interpreted scripting languages, such as Perl.
- *Platform Independence*—Servlets are platform-independent classes, so they can be run under different servers on different operating systems.
- *Consistency*—Servlets use a standard API (the Servlet API), so they enjoy the support of many Web servers.
- *Power*—Servlets can access any of the Java APIs. For example, a servlet can use JDBC to access a data store, or access remote objects such as EJBs over RMI.
- *Support*—The Servlet API exposes a number of classes that greatly simplify many of the tasks a server-side programmer must perform. For example, the Servlet API provides direct access to response and request information (you do not have to explicitly parse request data), and it provides support for state management through a Session object and classes to manipulate cookies.

One of the greatest facets of the servlet is its ability to interact with other J2EE components. Servlets can interact with other servlets or EJBs in just the same way as a JSP. For example, Figure 2.6 shows a client calling a servlet and then that servlet accessing an EJB.

FIGURE 2.6
The interactions between a client, servlet, and EJB.

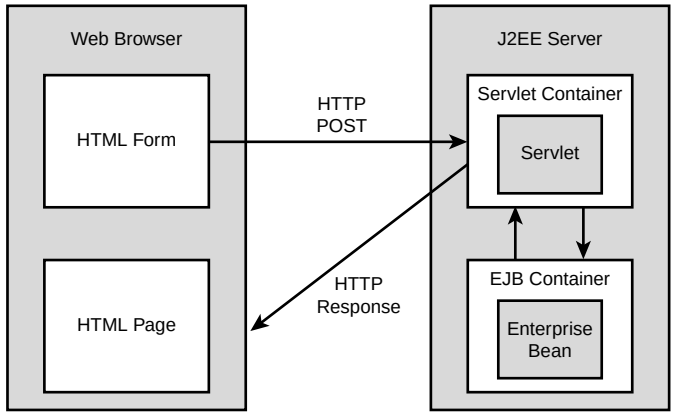


Figure 2.6 shows a user completing an HTML form and then initiating a HTTP `POST` to a servlet on a Web server. The Servlet Container on the Web Server invokes the servlet and passes it an object that represents the client's request. The servlet calls on the services of the EJB to perform the applications logic. Finally, the servlet generates an HTML response, which it returns to the client via a response object. In this instance, the servlet used an EJB, but it could also perform the processing itself or call another servlet or JavaBean.

Evaluation of Web-Centric Components

Now that you've been introduced to JSPs and servlets, you may wonder which component best suits a given scenario. In many instances, you can use JSPs and servlets interchangeably—remember that a servlet underlies a JSP. Both components dynamically create markup, operate on a request-response model, and can interact with other J2EE components.

However, Sun Microsystems provide guidance to help you develop applications using Web-centric components. This guidance takes the form of BluePrints that offer guidelines on the best practice and recommended use of J2EE technologies. The following guidelines derive from these BluePrints.

Generally, the presentation tier should use JSP pages that are presentation-centric, making them ideally suited to the generation of markup. In addition, JSP pages consist of

XML tags, which are familiar to Web content providers. This familiarity allows content providers to easily maintain a site's content without altering code. Conversely, you should consider servlets as a programmatic tool that you don't modify frequently. There are two main instances when you should use servlet in preference to JSPs:

- *Generating Binary Data*—You should use servlets to output binary data, such as images.
- *Extending Web Server Functionality*—For example, you could use a servlet to do the filtering of mail for a mail service.

As a guide, you should use JSPs in preference to servlets unless you require one of the previously mentioned items of servlet functionality.

The Client Tier

J2EE supports a wide range of clients. These clients range from thin clients, such as a Web browser, to intelligent clients, such as mobile devices that run J2ME Midlets. Both of these clients communicate through HTTP, but other clients may use SOAP, sockets, or even CORBA. The factors that unite these disparate clients are that they all call and subsequently receive a response from J2EE middle tier components. This part of the day looks at a range of J2EE clients and focuses on which components interact with them.

HTTP Clients

Java technologies have a long history of providing a wide range of services to clients that communicate via HTTP. For example, is it possible that someone has never heard of applets? J2EE provides services to Web-based clients by using two components—JSPs and servlets. These technologies may not be as familiar to you as the applet. Previously, this chapter explained what these components are; now, it explains how they integrate with HTTP clients.

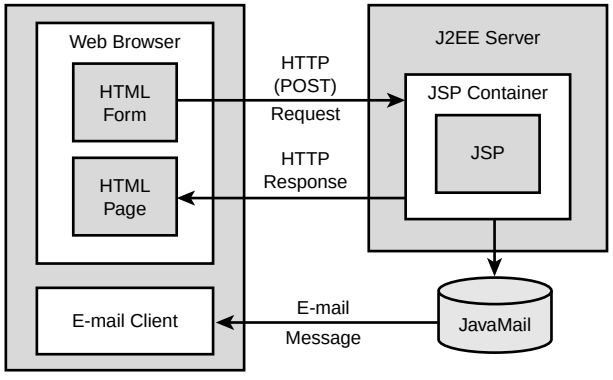
Static HTML Client

It's hard to imagine a Web application that relies on static HTML pages, but, believe it or not, some do. Typically, sites that use static pages are small and require very little functionality. The type of functionality they do require includes the processing of user response forms, basic e-commerce capabilities, and automated navigation. In a non-enterprise environment, a developer can use a CGI script to provide this functionality. With J2EE, you can use a servlet. For example, consider a customer feedback form written in HTML.

Figure 2.7 shows that the user completes the HTML Form and then initiates a HTTP POST to the servlet. The servlet parses the POST data; at this point, it could pass the

data to another component. However, in this instance, it simply uses classes in the JavaMail API (see Day 11, "JavaMail") to send the customer an e-mail response.

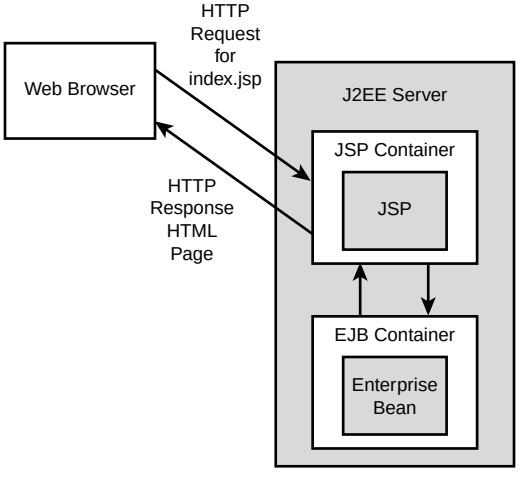
FIGURE 2.7
Using a servlet to process an HTML form.



Dynamic HTML Client

Many Web applications use HTML pages, which they generate when they receive a client request. These dynamic pages typically contain information that is context sensitive. For example, the page may contain a simple time stamp, a banner advertisement, or information retrieved from a database. With J2EE, you can use a JSP to create dynamic HTML pages. Figure 2.8 shows how JSPs relate to a Web client.

FIGURE 2.8
A Web-client interacting with a JSP.



The client makes a HTTP request for the page `index.jsp`. The JSP engine (on the server) interprets the tags within the JSP, calls the EJB, the EJB returns a response, and then the engine returns a page of HTML to the client. The functionality in this model is encapsulated in the EJB.

Java Applet Client

In case you do not know, an applet is a small GUI-based program that typically executes within the context of a Web browser. In the sphere of the Internet, a client requests an HTML page that references a server-side Java class file (the applet). The Web server responds to the client by returning this file. The applet then executes within the Java Virtual Machine (JVM) the client browser supplies.

In some respects, applets are an ideal way of providing remote access to J2EE applications. They are highly portable, run in an environment with strict security controls (the Java 1.0 Sandbox as a minimum), enjoy wide industry support (for example, Netscape and Internet Explorer browsers), and offer a rich GUI. However, browsers do not keep pace with changing specifications; so many browsers only support the deprecated Java 1.0 event model. This may cause you a number of problems, especially when working with AWT, which underwent a major change in its event model between Java 1.0 and Java 1.1. One of the best ways you can control the presentation tier is to deploy applets within a controlled network, such as a corporate intranet. In this instance, you can write code to work to the limitations of known browsers rather than working to the lowest common denominator.

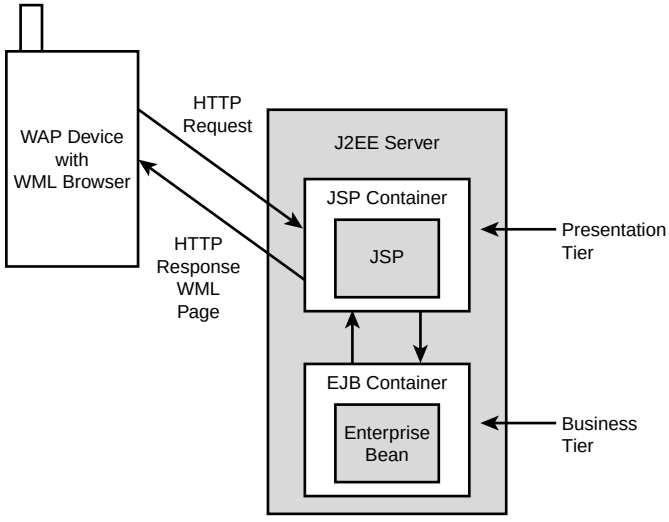
Other HTTP Clients

The three previous clients are very desktop browser centric, but there are many other types of clients that communicate by using HTTP. These clients include mobile devices, such as cellular phones, smart phones, and PDAs. For example, Figure 2.9 shows a WAP device calling a JSP. The component-based architecture means that the only change is to the JSP because WAP devices don't display HTML. All of the logic in business tier remains unaffected.

You can also write an application that uses HTTP to communicate and still communicate with the Web-centric J2EE components.

FIGURE 2.9

A WAP device with a WML browser interacting with a JSP.



Standalone Client

The HTTP clients all used a model that in essence was client—presentation tier—business tier—integration tier. However, there are other clients that may assume the responsibilities inherent in some of these tiers. For example, an application can connect to an EJB via its container, rather than route through a JSP in the presentation tier. Figure 2.10 shows such an example.

FIGURE 2.10

A standalone client directly interacting with an EJB.

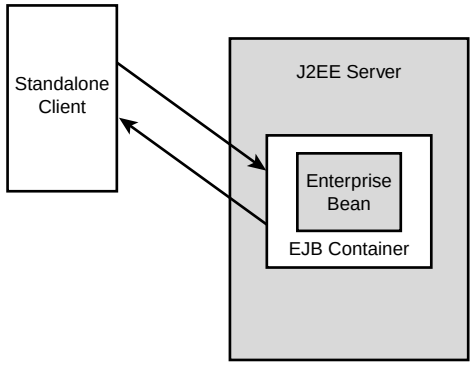
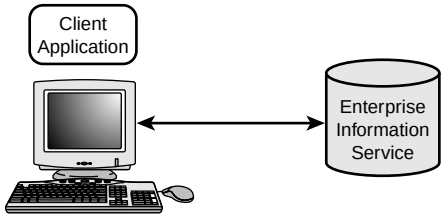


Figure 2.10 shows a standalone client accessing an EJB via its container. In this example, the client would typically be another EJB. Because the client accesses the business tier, it becomes responsible for the provision of the presentation tier. Consequently, in this example, the client EJB can pass the data to a servlet, which will then generate the appropriate response to its client.

Figure 2.11 shows another scenario where the standalone client bypasses both the presentation tier and business tier to access enterprise information system resources directly. Typically, the client uses JDBC to access the resources.

FIGURE 2.11

A standalone client interacting directly with an EIS resource.



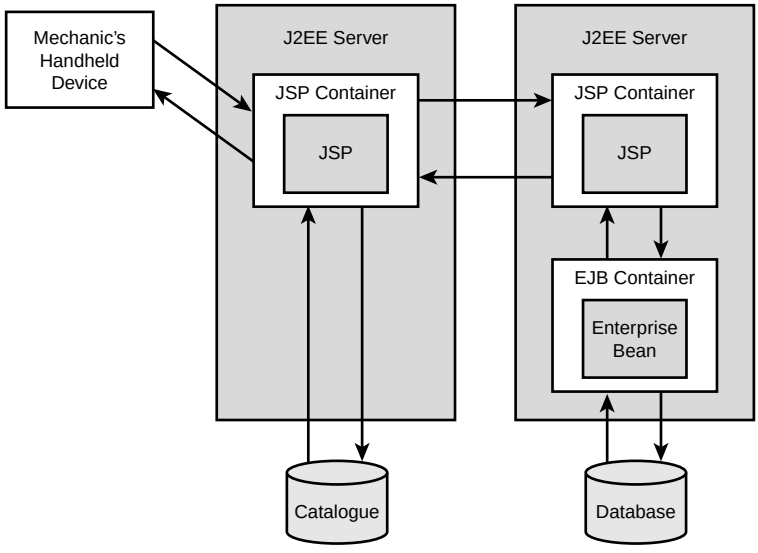
In this scenario, the client takes responsibility for the presentation tier and business tier. Interestingly, the client may not be another J2EE client: it might be a single application that encapsulates all presentation and business logic. If this is the case, the client will not gain the benefits of enterprise computing, especially scalability, because it is effectively working on the 2-tier client-server model.

Business to Business

The previous clients and scenarios worked on the premise of connecting to a component in a different tier—with the exception of the EJB to EJB example. However, many components do connect to components within the same level tier as themselves. For example, consider a small garage where mechanics order parts from a local catalogue via a handheld device. Figure 2.12 shows the architecture for the system. As you can see, a JSP handles their orders by accessing a catalogue via JDBC. With each order for a part, the JSP decrements the count for that part. When the part count reaches zero (Just-in-Time stock management), the JSP connects to a JSP at the parts wholesaler and places an order for more parts.

The previous example showed two JSPs communicating, but this could just as easily be two servlets or two EJBs. Another interesting development in peer-to-peer communications such as these is the use of messaging. For instance, the previous example requires both JSPs to be available at the same time to complete an order. However, using the messaging facilities JMS provides, you may create a system where both JSPs can asynchronously send and receive orders. You can learn a little more about JMS later today and a lot more on Day 9, “Java Message Service.”

FIGURE 2.12
JSP to JSP interaction.



Non-Java Clients

It is possible to use non-Java clients to access J2EE applications and application components. One obvious solution is that non-Java clients can use HTTP to access the services offered by a servlet or JSP. This is particularly attractive when the servlet or JSP produces and consumes data-oriented information in XML rather than HTML. This could be done in the context of exposing the component as a Web Service, which is discussed in the next section of today's lesson.

In the case of an EJB, things are a little trickier. However, the RMI-IIOP protocol used to communicate with EJBs allows them to interoperate with clients written using the Common Object Request Broker (CORBA) standard. Hence, it is possible to expose an EJB as a CORBA server that can be used by CORBA clients written in C++ or even COBOL.

Another alternative is that EJBs can be accessed from a Microsoft COM client using the J2EE Client Access Services (CAS) COM Bridge or Table Bridge.

Web Services

Web Services are XML-based middleware components that applications access over HTTP and SOAP. They enjoy industry-wide support and are not a proprietary solution. In fact, because Web Services use XML and open communication standards, any client that

can understand SOAP messages can consume Web Services. J2EE provides a rich framework that facilitates the building, deployment, and consumption of Web Services. You can learn more about J2EE and Web Services on Days 20 and 21.

Understanding Containers

When you previously installed J2EE, you also installed Sun Microsystems' reference implementation of J2EE, which is a J2EE Product. All J2EE products must provide containers to house J2EE components. The role of the container is to provide a component with the resources it needs to operate and a runtime within which to execute; yet still provide a degree of protection (security) to the application host.

Containers provide a number of services for a component. These services include lifecycle management, threading, security, deployment, and communication with other components. In addition to these services, a container must provide components with Java compatible runtime that conforms to J2SE 1.3.

Different components perform different tasks, so it may come as no surprise that they require different containers. A Product Provider can supply any of the following containers:

- Applet container
- Application client container
- JSP container
- Servlet container
- Web container
- EJB container

It was mentioned previously that all containers must supply a J2SE 1.3-compatible runtime environment. Interestingly, many applet hosts (typically Web browsers) do not support such a high runtime version. To compensate for this, the J2EE specification states that an applet container can use a Java plug-in to provide an appropriate environment.

All of these containers must provide the components they house with certain services and communications protocols. You will learn more about these service and protocols later in this chapter in the "Understanding the Service Supply to their Components" section. These containers must all also provide access to certain J2EE APIs; Table 2.2 shows these APIs.

TABLE 2.2 J2EE Required Standard Extension APIs

API	*Applet*	*Application Client*	*Web*	*EJB*
JDBC 2.0 Extension	N	Y	Y	Y
JTA 1.0	N	N	Y	Y
JNDI 1.2	N	Y	Y	Y
Servlet 2.2	N	N	Y	N
JSP 1.1	N	N	Y	N
EJB 1.1	N	Y	Y	Y
RMI-IIOP 1.0	N	Y	Y	Y
JMS 1.0	N	Y	Y	Y
JavaMail 1.1	N	N	Y	Y
JAF 1.0	N	N	Y	Y

For the sake of simplicity, Table 2.2 does not distinguish between servlet, JSP, and Web containers. You can consider these three containers as a stack where each builds on the functionality of the other. At the bottom of the stack is the servlet container, which must support HTTP; optionally, it might support other protocols. Above this is the JSP container, which provides the same functions as the servlet container and an engine to build servlets from JSP pages. Finally, the Web container provides all the services of the JSP container and access to J2EE service and communications APIs, which today's lesson details next.

Understanding the Services Containers Supply to Components

Previously, you learned that there are a variety of J2EE containers, and that these containers house J2EE components and provide methods and protocols that allow components to communicate with each other and with platform services. You also learned that a J2EE server underlies the container. This server is often known as the J2EE Product; there are other possible J2EE products that you will learn about later today in the "J2EE Product Provider" section of today's lesson.

J2EE products must provide components with certain standard services. Yesterday you were introduced to some of these services; today you will explore them in a little more detail and see how they work in conjunction with J2EE components.

Hypertext Transfer Protocol (HTTP)

A W3C specification defines HTTP 1.0, which is a protocol that allows the exchange of data of various formats in a widely distributed network. Both JSPs and servlets allow clients to access a J2EE application through the use of HTTP 1.0. Clients that have a Java runtime communicate with J2EE applications by using the `java.net` package, which is part of J2SE.

> The J2EE Specification states that containers need only provide support for HTTP 1.0. However, in practice, the majority of containers, including those in the RI, support HTTP 1.1.

HTTP over Secure Sockets Layer (HTTPS)

A Netscape specification defines SSL 3.0, which is a protocol that manages the secure transfer of data over a network. HTTPS uses SSL 3.0 as a sub-layer to HTTP to provide secure data transfer over the Internet. In common with HTTP, the JSP and Java Servlet APIs define the server-side API, and `java.net` defines the client-side.

Java Database Connectivity (JDBC)

JDBC is an API that allows you to access any tabular data source including relational databases, spreadsheets, and text files. The API allows you to connect to a database via a driver and then execute Structured Query Language (SQL) statements against that database. Appendix B, "SQL Reference," provides an SQL reference. The API consists of two packages—`java.sql` (ships with J2SE) and `javax.sql` (ships with J2EE). The `javax.sql` package provides many of the features an enterprise application requires, such as transaction support and connection pooling.

Day 8, "Transactions and Persistence," describes JDBC, so today's lesson provides only a quick overview of JDBC architecture. Typically, an EJB uses the API, but any other component can use it, for example a servlet. To connect to a given database, you must load a JDBC driver for that database. However, in the case of an Open Database Connectivity (ODBC) data source, you may optionally use a JDBC-ODBC bridge where no driver exists. After the appropriate driver loads, you can make a connection to the database and then create and execute SQL statements. If the statement is a query, a `ResultSet` is returned, which contains the results of the query. You can then manipulate these query results.

Java Transaction API (JTA)

A transaction is an atomic group of operations. For example, a banking application may debit one account and credit another. The transaction is considered complete when both the debit and credit are complete. If one operation fails, the other must roll back. A distributed system makes transaction management complex. In such a system, a transaction manager must coordinate transactions across the system.

The JTA API allows you to work with transactions independently of the transaction manager. You work directly with the methods JTA exposes via an instance of `UserTransaction`. In a simple scenario, you can use the `begin()`, `commit()` and `rollback()` methods—which might be familiar SQL commands if you are a database programmer—to manage the transaction. Day 8 explores JTA and its use with JDBC.

Java Authentication and Authorization Service (JAAS)

Anyone who has an interest in security knows that Java technologies have a rich history of providing a strong security framework. JAAS is a new supplement to this existing security framework. It provides both authorization and authentication services that the Pluggable Authentication Module (PAM) provides. In common with the Java 2 security framework, JAAS provides access control based on code location and code signers. In addition, JAAS provides access control to a specific user or group of users.

JAAS allows you to simply swap at runtime between encryption algorithms when authenticating users. This is because you interact with JAAS through a login context, so you effectively work with an abstraction of the authentication mechanisms.

Because JAAS does not actually contain classes that encrypt data, it is not subject to U.S. export control restrictions. This means that developers outside of the U.S. are free to download JAAS.

JAAS is an optional package to J2SE 1.3.x, and it ships with sample authentication modules that use JNDI, Solaris, and Windows NT. You can download the current version of JAAS from `http://java.sun.com/products/jaas/`. You can learn more about using JAAS with JNDI and J2EE components on Day 15, “Security.”

Java API for XML Parsing (JAXP)

XML is a text-based markup language that describes data. It provides a platform-independent and language-independent method for exchanging data between applications.

Because XML consists of plain text, it is human-readable. However, you very rarely read XML: you use an application that implements an XML API to read the data.

The JAXP API allows you to parse XML documents using the Document Object Model (DOM) or the Simple API for XML (SAX). One very useful feature of JAXP is that you can swap between XML parsers without making changes to your code. For example, if speed suddenly becomes very important, you can use the SAX parser because it reads a very large document in a fraction of the time of the DOM parser. Another useful feature of JAXP is that it provides support for Extensible Stylesheet Language Transformations (XSLT). For example, the J2EE Reference Implementation (RI) provides a transformation engine that supports XSLT. This allows you to dynamically transform an XML document into either another XML document, HTML, or plain text.

There are many ways that you might use XML within a J2EE application. For example, you can store content in XML and then transform the XML using XSLT so that a JSP can render content to devices that support different markup languages. Another typical use of XML is in the arena of business-to-business applications, where organizations can exchange data independently of their system architectures. Finally, one very important use of XML is within Web Services, which you will learn about on Days 20 and 21.

Java Naming and Directory Interface (JNDI)

JNDI provides an API for working with naming and directory services. A naming service simply associates names with objects, for example, the Domain Name System (DNS). A directory service also associates names with objects, but it also provides additional information through attributes, for example, a Lightweight Directory Access Protocol (LDAP) directory.

Although JNDI provides access to a wide array of naming and directory services, each service must provide a Service Provider. This is similar to JDBC and drivers, but in this instance, it is a naming or directory service and a Service Provider. For example, an LDAP directory must provide an LDAP Service Provider, which JNDI hides from you as a developer.

Whenever you need to access naming or directory services, you can use JNDI. More specifically, you use JNDI in J2EE in three main instances—to access or register EJBs or objects in an RMI registry or to access the CORBA Common Object Services (COS) naming service. You can learn more about JNDI in Day 4.

JavaBeans Activation Framework (JAF)

Typically, you use JAF in the context of JavaBeans (that's JavaBeans, not Enterprise JavaBeans!). However, a J2EE product must provide JAF for the JavaMail API to use

MIME types. JAF allows you to send e-mails that are not simply plain text. Instead, you can use different MIME types or send attachments. You can learn more about the use of JAF in the context of JavaMail on Day 11.

JavaMail

The JavaMail API provides classes that allow you to work with e-mail. Specifically, it allows you to send and receive e-mails by using a wide variety of protocols, including POP, SMTP, and IMAP. You can create e-mails that conform to a large number of MIME types, because the API uses JAF to provide support for a number of MIME types. For example, you can create HTML messages that contain embedded graphics and even have attachments.

Most Internet applications require the ability to send e-mail messages. You can use this API together with JAF to send e-mails from a JSP, a servlet, or an EJB. You can learn more about the API on Day 11.

Java Message Service (JMS)

Messaging is the process of communication between applications or components; it does not include application to human communications, such as e-mail. The JMS API allows you to create, read, and store messages.

JMS support two messaging models—point-to-point and publish-subscribe. Point-to-point messaging is where an application sends a message to a queue (a prearranged destination for messages), and then a client application collects that message. For example, leaving a voicemail message is a real-world example of this model. The publish-subscribe model requires client applications to subscribe to a topic with a message broker that again acts as a prearranged message destination. When an application sends a message to the message broker, the message broker immediately forwards the message to all current subscribers.

You can send and receive JMS messages by using both session and entity beans. However, you can only do this synchronously; the sender must suspend execution until the receiver receives the message. Alternatively, you can use a message-driven bean to send messages asynchronously. You can learn more about message-driven beans on Day 10 and JSM on Day 9.

Java Interface Definition Language (Java IDL)

Java IDL provides a way for you to access and deploy remote objects that comply with the Common Object Request Broker Architecture (CORBA) defined by the Object Management Group (OMG). CORBA Interface Definition Language (IDL) provides a

language-independent means of defining object interfaces. OMG provides mappings between various languages and IDL. A client written in any language that has an IDL binding can access objects you export by using CORBA. For example, a Java client can consume objects written in other languages, such as C++, C, Smalltalk, COBOL, and Ada.

In terms of J2EE applications, you can look up CORBA remote objects in the COS naming service through JNDI. The main reason you would want to use Java IDL, as a Java developer, is to allow your J2EE application to access legacy systems. For example, you might have a legacy Integration tier where a COBOL application manages data access. Java IDL allows an EJB in the business tier to communicate with the COBOL object, thus negating the need to rewrite the entire backend legacy code.

Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP)

RMI is a distributed object system that allows Java objects to interact with other Java objects running in a different virtual machine (usually on a remote host). In practice, you can access these remote objects almost as if they were local; you simply get a reference to the remote object and invoke its methods as if it were running in the same virtual machine. The seamless nature of this access is due, in part, to the fact that RMI is a Java-only distributed object system that relies on a proprietary transport protocol, namely, Java Remote Method Protocol (JRMP), to exchange information between client and server. However, this means that you can only use it to access other Java objects—not non-Java objects.

The actual process of performing a remote method invocation is similar to that of CORBA, namely, RMI utilizes client-side stubs and server-side skeletons. A client invokes a remote method by making a request on the stub, and this forwards to the server where the skeleton converts the request into an actual method call on the remote object. Any arguments for the remote method are marshaled by the stub into a serialized form before they are forwarded to the skeleton, which, in turn, unmarshalls the arguments. This marshalling allows objects to transport across a network.

Unfortunately, this protocol is not suitable for the type of enterprise-level interactions required by EJBs, where transaction and security context must be propagated across remote method calls. To this end, Sun Microsystems created a new implementation of RMI called RMI-IIOP. This keeps the same semantics as RMI (remote interfaces, passing serialized objects, and so on) but uses the CORBA Internet Inter-ORB Protocol (IIOP) as its transport mechanism. IIOP already contains all of the necessary hooks to propagate security and transaction context, so this new protocol can act as the core transport for EJBs in the J2EE architecture.

RMI-IIOP is used by default as the transport mechanism when generating stubs and skeletons for EJBs. You can also explicitly create RMI-IIOP clients and servers for your own applications by using flags on the RMI compiler (rmic) to create RMI-IIOP stubs and skeletons rather than the default JRMP stubs and skeletons.

You can also use RMI-IIOP as a mechanism for exposing your EJB components to CORBA clients without having to learn IDL. You can specify a flag to the RMI compiler that gets it to generate CORBA IDL on your behalf. After you have the CORBA IDL, this can be used together with an alternative language binding to create a client for the EJB that is written in another language.

Connector Architecture

The J2EE Connector Architecture allows your J2EE application to interact with Enterprise Information Systems (EIS), such as mainframe transaction processing, Enterprise Resource Planning (ERP) systems, and legacy non-Java applications. It does this by allowing EIS vendors to produce a resource adapter that product providers can plug into their application servers. The J2EE developer can then obtain connections to these EIS resources in a similar way to obtaining a JDBC connection.

The J2EE Connector Architecture defines a set of contracts that govern the relationship between the EIS and the application server. These contracts determine the interaction between server and EIS in terms of the management of connections, transactions, and security. You can learn more about Connector Architecture on Day 19, “Integrating with External Resources.”

Introducing Platform Roles

To create, package, and deploy any J2EE application—other than the simplest application—requires the effort of more than one person or organization. For example, in the development arena, a team of developers will write the J2EE components and someone else will assemble the finished application. In the production environment, someone will configure the J2EE environment and deploy the application, and yet another person will monitor the running application and its physical environment. In smaller organizations, there may be no physical distinction between these roles, but they will still be logically separate. Based on this premise, it is no surprise that Sun Microsystems suggest a named team whose responsibility it is to perform these tasks.

This team, together with Product Providers and Tool Providers constitute the J2EE platform roles. It is these roles that this section explores.

J2EE Product Provider

A J2EE product must include the component containers and J2EE APIs the J2EE specification states; today's lesson has introduced all of these containers and APIs. Examples of J2EE products include operating systems, database systems, application servers, and Web servers. An organization that supplies a J2EE product is known as the J2EE Product Provider.

The J2EE Product Provider is also responsible for mapping application components to the network protocols the J2EE specification defines. In addition, the Product Provider must provide deployment tools for the Deployer and management tools for the System Administrator. The end of this section provides an explanation of these tools.

The Product Provider is free to provide implementation-specific interfaces that the J2EE specification does not define. Hence, you will occasionally see a warning in a lesson that highlights a vendor-specific piece of functionality.

Application Component Provider

As you have already seen, a J2EE application consists of components, but it also may consist of other resources, such as HTML files or XML files. The Application Component Provider creates both these resources and components. Almost all organizations will use several component providers. They may exist in-house, or the organization may outsource component creation or buy in components. Whichever is the case, specialists in the different tiers (presentation, business, and data access) will write the components that relate to that tier. For example, a business tier specialist will write EJBs, whereas a presentation tier expert may write JSPs. Regardless of the Application Component Provider's specialist area, the Tools Provider will supply them with tools to write components.

Application Assembler

After the Application Component Providers write an application, the Application Assembler assembles the application into a J2EE application. The Application Assembler packages the application into an EAR file that must conform to the J2EE specification. Other than assembly, the Application Assembler is responsible for providing instructions that state the external dependencies of the application. Typically, the Application Assembler uses tools the Tools Provider or Product Provider provides to perform these tasks.

Application Deployer

The Application Deployer is the first person who requires knowledge of the production environment. This is because he or she must deploy the application into that environment. Specifically, the Application Deployer must install, configure, and start the execution of the application. Typically, the Product Provider provides tools that help perform these tasks.

The installation process is where the Application Deployer moves the application to the server and installs any classes the container requires to perform its duties. During the configuration process, the Application Deployer satisfies any external dependencies the Application Assembler stipulates and configures any local security settings, for example, modifies a policy file. The final stage, starting execution, is where the Application Deployer starts the application in readiness to service clients.

Systems Administrator

The Systems Administrator configures and maintains the enterprise network, and monitors and maintains the application the Application Deployer deployed. The Product Provider supplies tools that assist the Systems Administrator in the monitoring and maintenance of the application.

This concludes the list of platform roles that specifically work with the application; only the Tool Provider remains. Figure 2.13 shows the interactions between each of these roles and their interactions with the J2EE application.

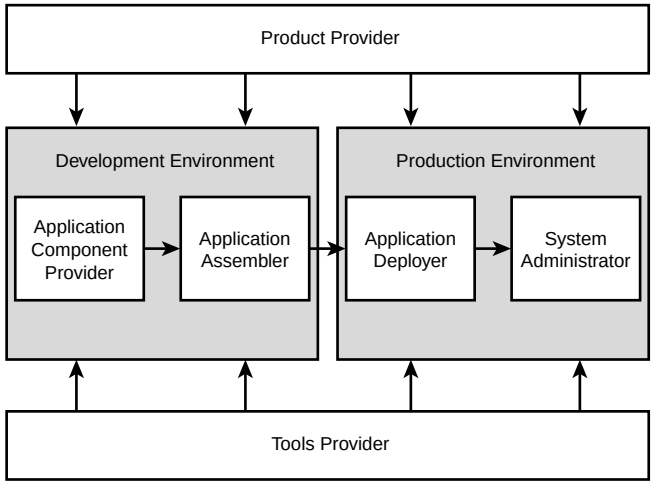


FIGURE 2.13
J2EE roles.

Tool Provider

As you have seen, many of the platform roles use tools the Tool Provider supplies. These tools assist people with the creation, packaging, deployment, and maintenance of J2EE applications. Currently, the J2EE specification only defines that the Product Provider must supply deployment and maintenance tools; it does not stipulate what these tools should be. Future releases of the specification are likely to provide further guidelines, so that Tool Providers can supply platform-independent standardized tools sets.

To offer you a typical overview of the tools that Tools Providers and Product Providers supply, this section concludes with a brief survey of the tools that ship with the J2EE reference implementation. For use guidance, please refer to the J2EE documentation.

- *J2EE Administration Tool*—Enables the addition and removal of resources, such as JDBC drivers and data sources.
- *Cleanup Tool*—Removes all J2EE applications from the server.
- *Cloudscape Server*—Starts and stops the Cloudscape relational database.
- *Deployment Tool*—Enables you to package and deploy J2EE applications.
- *J2EE Server*—Launches and stops the J2EE server.
- *Key Tool*—Enables you to generate public and private keys and X.509 certificates.
- *Packager*—Allows you to package J2EE applications if you are not packaging them using the deployment tool (see above). You can create EJB JAR, Web WAR, Application Client JAR, J2EE EAR, and Resource Adapter RAR files.
- *Realm Tool*—Allows the administration of J2EE users and also the import certificates.
- *Runclient script*—Enables you to run a J2EE application client.
- *Verifier*—Verifies the integrity of EAR, WAR, and JAR files.

Future of J2EE tools

There are 3 Java Specification Requests underway that will affect the future of J2EE tools:

- JSR 77—A new management model for tools that will provide a single management tool to configure the J2EE platform. You can read more about the JSR at `http://www.jcp.org/jsr/detail/077.jsp`.
- JSR 88—A description of the APIs that enable the deployment tool. You can read more about the JSR at `http://www.jcp.org/jsr/detail/088.jsp`.

- JSR 127—This defines the architecture that simplifies the creation and maintenance of Java Server application graphical user interfaces. You can read more about the JSR at `http://www.jcp.org/jsr/detail/127.jsp`.

Packaging and Deploying J2EE Applications

The installation of a typical desktop application, such as a word processor, is usually a straightforward affair. The installation program will ask you a few questions about the functionality you require and where it should install its files. It will also examine parts of your desktop machine (such as the Windows registry) to discover whether any components that it relies on are already installed. Normally, such an installation takes in the order of a few minutes—half an hour at most.

The installation of a distributed enterprise application is unlike that of a packaged desktop application. The installation of a desktop application is reasonably straightforward because

- The concept of word processing is well understood by most people, so they can make an appropriate judgment on whether they need particular parts of the package or not. There is little in the way of personal tailoring involved.
- All of the installation takes place on a single machine. The installation program knows where to find existing configuration information and where it should install the different parts of the application.

Compared with this, a distributed enterprise application will require a lot more information about the environment in which it is to be installed. This includes, but is not limited to, the following:

- The location of the servers on which the server-side components will be deployed. The Web and business components for an application could be distributed across multiple servers.
- The appropriate level of security must be enforced for the application. The application must carry with it information about the security roles it expects and the access each role has to the functionality of the application. These security roles must be mapped onto the underlying security principals used in the distributed environment.
- Components that access data and other resources must be configured to use appropriate local data sources.

- The names of components and resources must be checked and potentially changed to avoid clashes with existing applications or to conform to a company-wide naming standard.
- Web-components must be configured so that they integrate with any existing Web sites of which they will form a part.

As you can see, this is currently a far more specialist task, requiring knowledge about the application and the environment in which it is being deployed. The application must carry with it information about the requirements it has of the environment. The application assembler defines these requirements when the application is created. The application Deployer must examine these requirements and map them onto the underlying environment.

J2EE applications

A J2EE application will consist of the following:

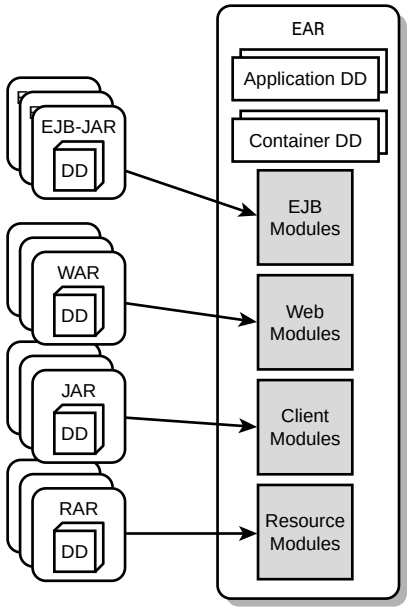
- Zero or more Web components packaged as Web Archives (WAR files)
- Zero or more EJB components packaged as EJB-JAR files
- Zero or more client components packaged as JAR files
- Zero or more connectors packaged as Resource Archives (RAR files)

Naturally, there must be at least one component for there to be an application! The components that constitute an application must be packaged together so that they can be transported and then deployed. To this end, all of the components in a J2EE application are stored in a particular type of JAR file called an *Enterprise Application Archive* or *EAR*.

Given the previous scenario, it should be clear that a J2EE application needs to carry with it information about how its different parts interrelate and its requirements of the environment in which it will be deployed. This information is carried in a series of XML documents called *deployment descriptors*. There is an overall application deployment descriptor that defines application-level requirements. This application deployment descriptor is also stored in the EAR file.

Each individual component will have its own deployment descriptor that defines its own configuration and requirements. These component deployment descriptors are carried in the individual component archives described in the “Breaking Modules Down into Components” section of today’s lesson. Figure 2.14 shows the structure of an EAR file and how the application deployment descriptor, the component archives, and the component deployment descriptors fit within this structure.

FIGURE 2.14
Structure of an Enterprise Archive (EAR).



The application deployment descriptor contains application-wide deployment information and can potentially supersede information in individual component deployment descriptors.

Note

The EAR file can also contain a container-specific deployment descriptor that holds information that is useful to the container but falls outside the scope of the J2EE application deployment descriptor.

The application is split into modules, each of which represents a component. If necessary, a module can contain an additional deployment descriptor to override some or all settings in the deployment descriptor provided in the component archive file.

Breaking Modules down into Components

As you can see from Figure 2.14, components are represented in an EAR file by component archive files. Each module will point to its associated component archive file. Each type of component archive file is a JAR-format file that contains the component's classes and resources together with a component-specific deployment descriptor.

The two most common types of component archive are EJB-JAR files and WAR files.

EJB Component

An EJB-JAR file contains all of the classes that make up an EJB. It also contains any resource files required by the EJB. The properties of the component are described in its associated deployment descriptor, called `ejb-jar.xml`, which is also included in the EJB-JAR file.

The deployment descriptor described the main class files contained in the EJB-JAR file. The deployment descriptor specifies which external resources are required by the component. It also contains extra information about the security and transaction settings. This resource and extra information is often referred to as metadata. Figure 2.15 shows a subset of the contents of an EJB deployment descriptor.

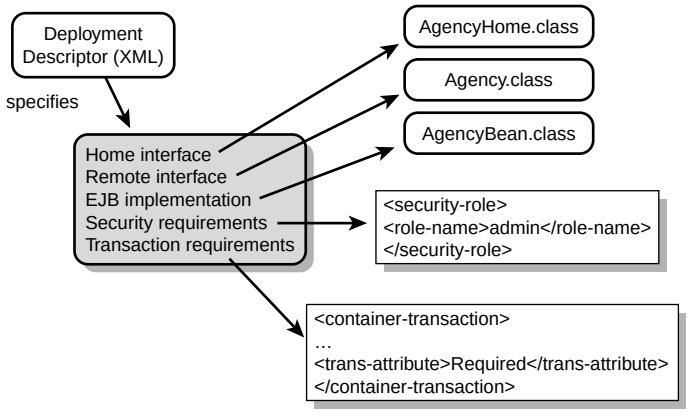


FIGURE 2.15

An EJB deployment descriptor indicates the main classes in the EJB-JAR file together with the component's metadata.

All of the component's metadata can be altered or replaced by the application assembler as they bind the component into the application. The application Deployer can also customize some of the metadata.

An EJB-JAR file can contain more than one EJB.

EJB-JAR files and their deployment descriptors are discussed in more detail on Days 4 and 5. Other aspects of EJB deployment information, such as security and transactions, are covered later.

Web Component

Servlets and JSPs can also be packaged together into a component archive file. The archive file is a JAR-format file that contains the class files, JSP files, and resources required by the Web component. In this case, the resources can include static HTML files that form part of the application. This Web Archive (WAR) file also contains a deployment descriptor that indicates the Web components contained in the WAR.

Just as with the EJB deployment descriptor, the WAR deployment descriptor indicates the main classes in the WAR file and the resources required by the components. However, the WAR deployment descriptor also contains Web-specific information, such as the URLs onto which servlets and JSPs should be mapped, and which is the front page of the application.

WAR files and their deployment descriptors are discussed in more detail on Day 12, "Servlets," and Day 13. Other aspects of WAR deployment information, such as security, are covered later.

Summary

Today, you looked in more detail at J2EE and the facilities that it provides. You saw how the different J2EE technologies fit into the 3-tier model, and how it provides component frameworks for different types of functionality.

You have seen that the EJBs provide a robust, scalable home for business logic and that servlets (and JSPs) provide a flexible way of delivering application functionality to clients. There are many different types of client, ranging from simple, markup-based clients that work over HTTP to sophisticated and powerful clients that use GUIs and RPC.

You have seen that the creation and deployment of an enterprise application requires many different roles. You have also seen that an enterprise application is assembled from many different parts, and that it must carry with it information about how all of those parts fit together.

Q&A

Q Can a J2EE application be written without using any Enterprise JavaBeans?

A Certainly. You can write a client application that connects to a servlet in a Web container and have that servlet connect directly to a back-end database. You don't

need to add an EJB. An EJB can add value by providing persistent conversational state if that is required. It can also provide transactional security and roll back to a previous state should there be an interruption in the flow of data for any reason. Therefore, you can use servlets and JSPs on their own if a database is simply read, but any updates or new records to be added will more safely be done using Enterprise JavaBeans.

Q What type of EJB should I typically use to encapsulate business logic? And which type would I use to contain data and its associated operations?

A For pure business logic, you would typically use a session bean (or a message-driven bean). If the EJB is to represent underlying application data, you would probably use an entity bean.

Q Why are JSPs generally faster than other server-side scripting environments?

A When a JSP is accessed, it is compiled into Java bytecodes. Every subsequent access will use the bytecodes rather than processing the page again. This means that JSPs will run as fast as standard Java classes such as servlets.

Q What are the consequences of producing a J2EE application with vendor specific APIs?

A The application you produce will become specific to a particular container, server, or vendor. You will not be able to easily move the application from platform to platform.

Q How do you package an EJB? What should be in the package?

A An EJB is packaged in an EJB-JAR file. The EJB-JAR file contains the classes for the EJB, any other resources, and a deployment descriptor that contains EJB metadata and describes the external resource requirements of the EJB.

Q What is an EAR file?

A This is an Enterprise Application Resource file that houses the application's JAR, WAR, and XML files. The Assembler takes on the responsibility of packaging the EAR file, while the Deployer authenticates that the file conforms to the J2EE specification, adds the file to the J2EE server, and deploys the application.

Exercises—Case Study

To help understand the role of each of the technologies in the J2EE specification, a single case study will be followed throughout the daily exercises. As you work through the 21 days, a functional implementation of a simple enterprise application will be developed.

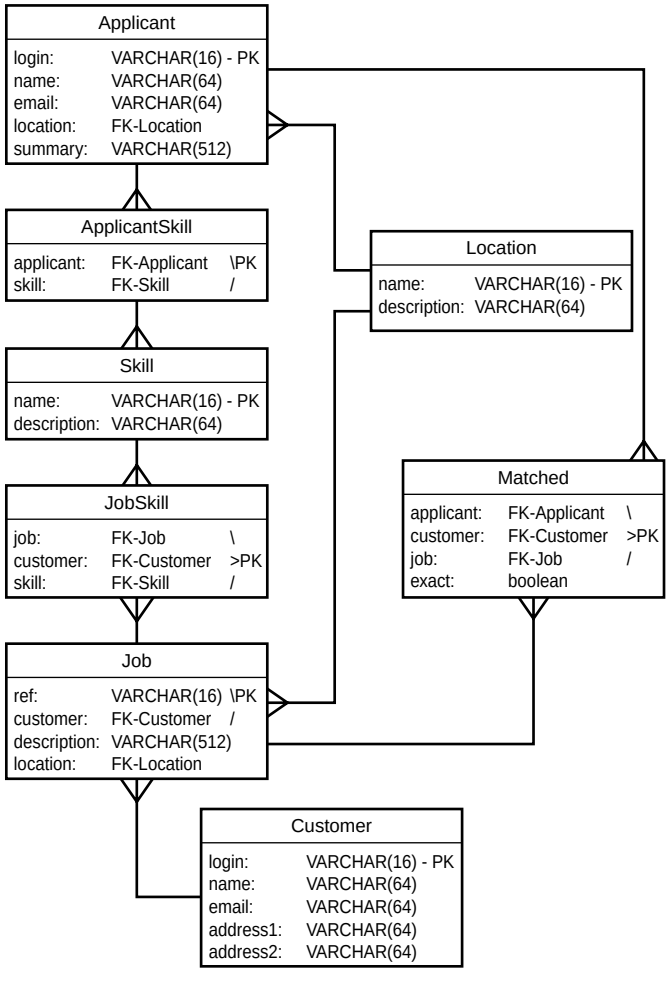
The Job Agency

The chosen case study is a simple Job Agency. Jobs are categorized by Location and Skills required for the job. Customers advertise jobs, and Applicants register their location and skills so they can be matched to jobs. Customers and applicants will be notified of job matches by e-mail.

To illustrate the relationships between the different components in the data model for the Agency, a traditional database ERD diagram is shown in Figure 2.16.

FIGURE 2.16

Case Study ERD.



Today's exercise, which is described later, will be to create the database and register it as a `javax.sql.DataSource` to the J2EE RI.

The case study has a front office part with the following components:

- Maintaining the location and job lookup tables
- Adding customers and advertising jobs
- Registering job applications

The back office part consists of the following:

- Matching a new job against existing applicants
- Matching a new applicant against existing jobs
- Generating e-mails

Using the Agency Case Study

The example code shown in each day's lesson will use the Customer functionality (jobs and invoices) from the case study. At the end of each day's work, you will be asked to enhance the case study by adding the Applicant functionality (registering jobs) to the system. A fully-worked solution for the exercises is provided on the CD-ROM included with this book, so that you will have a working case study if you choose to omit any day's exercise.

Material for many days exercises, particularly JNDI, JavaMail, JMS and Java Connectors, will primarily use examples and exercises not related to the case study. The last two days' work discuss J2EE in the context of the wider context of Web applications and do not refer to the Agency case study.

Table 2.3 shows roughly what will be covered on each day. Don't worry if some of the terminology is new to you; all will become clear as you work your way through the book.

TABLE 2.3 Daily Workout

Day	*Lesson*	*Exercise*
1	Introduce multi-tiered application architectures	No exercise
2	Introduce the J2EE platform, technologies and roles	Install J2EE RI and case study database.
3	Using JNDI naming and directory services	Write a JNDI namespace browser.

TABLE 2.3 Continued

Day	*Lesson*	*Exercise*
4	Using data sources, environment entries, and EJB references	Build and deploy a simple EJB and client application with J2EE RI.
5	Using Session EJBs to implement business logic	Add a Session bean to register job applicants.
6	Using Entity EJBs to encapsulate access to persistent data	Add Entity beans for the appli cant data and refactor the register Session bean.
7	Using Container Managed Persistence (CMP) and Container Manage Relationships (CMR) with entity EJBs	Refactor the applicant Entity bean to use CMP.
8	Adding transaction management to Session and Entity EJBs	Add transaction management to the Applicant processing.
9	Using JMS topics and queues	Develop a simple chat room service.
10	Using Message-driven beans to implement back office functionality	Use Message-driven beans to match new or changed applicants to advertised jobs.
11	Adding e-mail capabilities to back office functionality	Use e-mail to send matched jobs to applicants.
12	Developing Web-based applications using servlets	Develop a servlet front end to create a new applicant for the Agency case study.
13	Developing Web-based applications using Java Server Pages	Use JSPs to register job applicants.
14	Using custom Tag Libraries with JSPs	Refactor the register job JSP to use Tag Libraries.
15	Adding security to restrict access to J2EE application functionality and data	Add security to control access to the job skills data.
16	Understanding XML and writing simple XML documents	Refactor the messages sent to the back office job/applicant matching functionality to use XML.
17	Using XSL to transform XML documents into different data formats	Transforming an XLD document into HTML for viewing in a Web browser.

TABLE 2.3 Continued

Day	*Lesson*	*Exercise*
18	Understanding design patterns and recognizing patterns present (and absent) from the case study	Identify which design patterns can be applied to the case study to improve maintainability.
19	Working with legacy systems using the Connector architecture	Identify how the case study could be linked into a legacy invoicing system.
20	Exposing J2EE components as Web Services	Create a simple Java stock quote class and expose it as a Web Service. Create a client to retrieve the stock quote information from the server.
21	Using XML-based registries and asynchronous Web Services	Create a Web Service `JobPortal` that will take a SOAP message containing new agency customer information and return a generated customer login ID.

By the end of the course, you will have a simple, but working, job agency enterprise application. The agency will have both a GUI-based front end and a Web-based interface and will have given you a good grounding in the relative strengths of each J2EE technologies and how to apply them.

Practice Makes Perfect

Developing J2EE architectures requires two disciplines:

- Good analysis and design skills
- Practical hands-on experience with the J2EE technologies

The first comes with time and experience, but the last few days lessons will help point you in the right direction to becoming a J2EE designer.

The second discipline comes with practice. If you read this book and attempt all the exercises, you will learn a lot more than if you just read the book and simply study the example code shown.

The case study exercises are not complex. They have been designed to take between 30 minutes and 2 hours to complete. The exercises only use information presented and your existing knowledge: you will need to know something about JDBC, Swing, and HTML,

but you certainly don't need to be an expert in these technologies. The book "Teach Yourself Java in 21 Days" from SAMS Publishing is a good source for improving your knowledge of JDBC and Swing should you require a little refresher course. More importantly, the exercises will give you hands-on coding experience using J2EE.

The Case Study Directory on the CD-ROM

On the CD-ROM included with this book, you will find all the Java software required to develop all of the code shown in this book.

The CD-ROM contains a directory called `CaseStudy`. All the example code solutions to the exercises are included in this directory.

There are 21 sub-directories corresponding to each days work. Each day will have one or more of the following directories:

- `Agency` The complete Agency case study so far. This includes code from the examples in the book and the completed exercise if this is based on the case study.
- `Examples` The code for all the example programs show in the book where these examples are not part of the case study.
- `Exercise` Any existing code to be used as a starting point for the exercise. Typically, this will be the Agency case study, including all the example code in the book but excluding the code the reader needs to provide as part of the exercise.
- `Solution` A solution to the set problem if the exercise does not enhance the Agency case study.

Installing the Case Study Database

The Job Agency case study requires a small database for storing information about customers, jobs, applicants, and invoices. A Java program to create the database has been provided in the Day 2 exercises on the accompanying CD-ROM. The program uses the Cloudscape database provided with the J2EE RI and can easily be adapted to work with any JDBC compatible database.

Find the directory on the CD called `CaseStudy\Day02\Exercise`.

Inside this directory is a Java source file, class file, and two script files:

- `CreateAgency.java` A source file for a program to create the `Agency` database under J2EE RI Cloudscape database.
- `CreateAgency.class` The compiled Java class file for `CreateAgency.java`

- `CreateAgency.bat` A Windows NT/2000 batch file to run the application to create the database
- `CreateAgency.sh` Unix/Linux Bourne shell script to run the application to create the database

To create the Agency database, you will need write permission to the J2EE installation directory.

Follow the instructions shown earlier in today's lesson for stopping the Cloudscape and J2EE, servers and stop these servers if they are currently running. If you have installed J2EE as suggested, you simply have to enter the following commands from a command (or shell) window:

```
j2ee –stop
cloudscape -stop
```

The Java `CreateAgency` program provided in the Day 2 exercises will create the necessary database files in a sub-directory called `Agency`. To create and install the database, you will need to do the following:

1. Copy all the files from the Day 2 exercises directory CD-ROM to the `cloudscape` sub-directory of the J2EE SDK installation directory.
2. Change directories to the `cloudscape` sub-directory of the J2EE SDK home directory and then run the appropriate script program as follows:

   Under Windows, type

   ```
   cd %J2EE_HOME%\cloudscape
   CreateAgency
   ```

   Under Linux/Unix, type

   ```
   cd $J2EE_HOME/cloudscape
   ./CreateAgency.sh
   ```

   A new sub-directory called `Agency` will be created under the current `cloudscape` directory.
3. Having created the database, you must now add a data source called `Agency` to J2EE (data sources are discussed on Day 4, "Introduction to EJBs"). Run the following command to add the data source (the same command is used for both windows and Linux/Unix):

   ```
   j2eeadmin -addJdbcDatasource jdbc/Agency
   ➥jdbc:cloudscape:rmi:Agency;create=true
   ```

   If you have not included the J2EE bin directory in your program search path, you will have to run the command as shown below:

Under Windows, enter the following command:

```
%J2EE_HOME%\bin\j2eeadmin -addJdbcDatasource
➥jdbc/Agency jdbc:cloudscape:rmi:Agency;create=true
```

Under Linux/Unix, enter the following command:

```
$J2EE_HOME/bin/j2eeadmin -addJdbcDatasource
➥jdbc/Agency jdbc:cloudscape:rmi:Agency;create=true
```

Finally, restart the Cloudscape and J2EE servers as described in the earlier section, "Starting the J2EE Reference Implementation (RI)." Normally, you would start the J2EE server and Cloudscape database server in separate windows by using the following commands:

```
j2ee –verbose
cloudscape -start
```

If you start the J2EE server as shown with the `-verbose` option, you will see the diagnostic output shown in Listing 2.4 (the line showing the `Agency` data source configuration is highlighted in bold).

LISTING 2.4 The J2EE Reference Implementation Startup Diagnostics

```
   1: > j2ee -verbose
   2: J2EE server listen port: 1050
   3: Naming service started:1050
  4: Binding DataSource, name = jdbc/DB2, url =
➥jdbc:cloudscape:rmi:CloudscapeDB;create=true
   5: Binding DataSource, name = jdbc/DB1, url =
➥jdbc:cloudscape:rmi:CloudscapeDB;create=true
   6: Binding DataSource, name = jdbc/Agency, url =
➥jdbc:cloudscape:rmi:Agency;create=true
   7: Binding DataSource, name = jdbc/InventoryDB, url =
➥jdbc:cloudscape:rmi:CloudscapeDB;create=true
   8: Binding DataSource, name = jdbc/Cloudscape, url =
➥jdbc:cloudscape:rmi:CloudscapeDB;create=true
   9: Binding DataSource, name = jdbc/EstoreDB, url =
➥jdbc:cloudscape:rmi:CloudscapeDB;create=true
  10: Binding DataSource, name = jdbc/XACloudscape, url = jdbc/XACloudscape__xa
  11: Binding DataSource, name = jdbc/XACloudscape__xa,
➥dataSource = COM.cloudscape.core.RemoteXaDataSource@653220
  12: Starting JMS service...
  13: Initialization complete - waiting for client requests
  14: Binding: < JMS Destination : jms/Queue , javax.jms.Queue >
  15: Binding: < JMS Destination : jms/firstQueue , javax.jms.Queue >
  16: Binding: < JMS Destination : jms/Topic , javax.jms.Topic >
  17: Binding: < JMS Cnx Factory :
➥QueueConnectionFactory , Queue , No properties >
  18: Binding: < JMS Cnx Factory :
➥jms/QueueConnectionFactory , Queue , No properties >
```

LISTING 2.4 Continued

```
  19: Binding: < JMS Cnx Factory :
➥TopicConnectionFactory , Topic , No properties >
  20: Binding: < JMS Cnx Factory :
➥jms/TopicConnectionFactory , Topic , No properties >
  21: Starting web service at port: 8000
  22: Starting secure web service at port: 7000
  23: J2EE SDK/1.3
  24: Starting web service at port: 9191
  25: J2EE SDK/1.3
  26: J2EE server startup complete.
```


You will test the `Agency` database configuration on Day 4 when you learn how to create and deploy a simple EJB.

Congratulations, you have installed J2EE successfully and completed today's exercise to configure the Agency database for use with the other exercises in this book.

WEEK 1

DAY 3

Naming and Directory Services

The previous days have discussed the background to enterprise computing concepts and introduced J2EE technologies such as EJBs and Servlets. This chapter will show how the Java Naming and Directory Interface (JNDI) supports the use of many of the J2EE components.

In its simplest form, JNDI is used to find resources (such as EJBs) you have registered via the J2EE server. Advanced use of JNDI supports sophisticated storage and retrieval of Java objects and other information.

This day's work will include

- Using Naming and Directory Services
- JNDI and X.500 names
- Obtaining a JNDI Initial Context
- Binding and looking up names
- Name attributes
- Objects and References
- JNDI events and security

Naming and Directory Services

A *Naming Service* provides a mechanism for giving names to objects so that you can retrieve and use those objects without knowing the location of the object. Objects can be located on any machine accessible from your network, not necessarily the local workstation.

A real-world example is a phone directory. It stores telephone numbers against names and addresses. To find someone's phone number is simply a matter of using his or her name (and possibly address) to identify the entry in the phone book and obtain the stored phone number. There are a few complications, such as finding the right phone book to look in, but it is essentially fairly simple.

Incidentally, naming services have a similar problem to that of finding the right phone book. This is known as *obtaining a context*. A name can only be found if you examine the right context (phone book).

A *Directory Service* also associates names with objects but provides additional information by associating attributes with the objects.

The yellow pages phone directory is a simple form of a directory service. Here, businesses often include advertisements with additional information such as a list of products sold, professional qualifications, affiliated organizations, and even location maps for their premises. These attributes add value to the name entry. A directory service will normally provide the ability to find entries that have particular attributes or values for attributes. This is similar to searching the yellow pages phone book for all plumbers running a 24-hour emergency service within a certain area.

Yellow page style phone books also store names under different categories—for example, plumbers or lawyers. Categorizing entries can simplify searching for a particular type of entry. These categorized entries are a form of sub-context within the directory context of the local phone book.

Why Use a Naming Service?

Naming Services provide an indispensable mechanism for de-coupling the provider of a service from the consumer of the service. Naming services allow a supplier of a service to register their service against a name. Users, or clients, of the service need only know the name of the service to use it.

Think of the phone book once more, and how difficult it would be to find someone's phone number without the phone book. Obtaining your friend's phone number means going to their home and asking, or waiting until you meet up with them again—which may be difficult to organize because you can't phone them to arrange the meeting.

The phone book is a directory service. In fact, a phone book is often referred to as a phone directory. The phone directory service lets you look up a person or company's phone book using their name as a key.

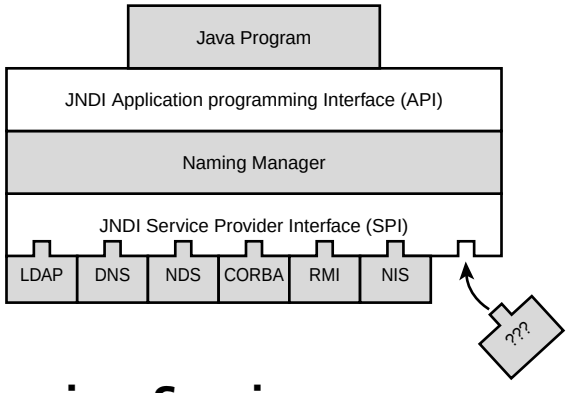
At the end of the day, it is very difficult to imagine a world without naming services.

What is JNDI?

JNDI is a Java API that defines an interface to Naming and Directory Services for Java programs. JNDI is just an API and not, in itself, a Naming and Directory Service. To use JNDI, an implementation of a Naming and Directory service must be available. JNDI provides a service-independent interface to the underlying Service Provider implementation.

Figure 3.1 shows how the JNDI layer interfaces between the Java program and the underlying naming services. Additional naming services can be plugged into the JNDI layer by implementing the Service Provider Interface (SPI) for JNDI.

FIGURE 3.1
JNDI Architecture.



Common Naming Services

Figure 3.1 shows that JNDI supports several well-known naming services, including the following:

- Domain Name System (DNS) is the Internet naming service for identifying machines on a network.
- Novell Directory Services (NDS) from Novell provides information about network services, such as files and printers. NDS is found primarily in environments where the main networking software is Novell.
- Network Information Service (NIS) from Sun Microsystems provides system-wide information about machines, files, users, printers, and networks. NIS is primarily found on Solaris systems, but Linux and some other Unix platforms support it.

- Lightweight Directory Access Protocol (LDAP) is the approved standard for an Internet naming service. LDAP is a true directory service and supports attributes as well as names for objects. LDAP is fast becoming the de-facto directory service for the enterprise.

JNDI also supports some more specialized naming systems. For example, CORBA for distributed component programming and RMI for distributed Java programming.

Although there is no named service provider for Windows Active Directory, it is supported. Windows Active Directory supports an LDAP interface, and you can access it via the JNDI LDAP Service Provider Interface.

Naming Conventions

Each naming service has its own mechanism for supplying a name. Perhaps the most familiar naming convention is that of DNS, where every machine connected to the Internet has a unique name and address. Most readers should recognize the following as a host name used by DNS:

```
www.samspublishing.com
```

In contrast, LDAP names are based on the X.500 standard and use distinguished names that look like the following fictitious example:

```
cn=Martin Bond, ou=Authors, o=SAMS, c=us
```

This format will also be familiar to users of Microsoft's Active Directory service, whose naming system is also based on X.500 but uses a forward slash to separate the various name components:

```
cn=Martin Bond/ou=Authors/o=SAMS/c=us
```

These last two naming conventions have similarities in that they are both hierarchically structured with the most specific name occurring first and the most general name (or context) occurring last.

JNDI provides classes that support creating and manipulating structured names; but most programmers will use simple strings that JNDI passes on to the underlying service with minimal interpretation.

Some JNDI Service Providers may use names that are case sensitive, and some service providers may not, it all depends on the underlying technology and environment. To maintain portability of your applications, it is always best to avoid names that differ only by letter case and also ensure that names are always spelled in a consistent manner.

Using JNDI

JNDI is a standard component of JDK 1.3 and is, therefore, also part of J2EE 1.3. JNDI is also included in J2EE 1.2 and is available as a standard Java extension for JDK 1.2 and earlier.

While developing code, the program's `CLASSPATH` must include the location of the JNDI class libraries. As long as the `JAVA_HOME` environment variable has been set up, the JNDI classes will be available to the Java compiler.

Running a JNDI-aware program requires a JNDI service to be running and the classes for that service to be available to the program. Typically, this requires the `CLASSPATH` to include one or more JAR files provided by the JNDI provider or a J2EE server vendor. For implementation-specific details, see the vendor's documentation.

By default, running a J2EE server starts a naming service on the same machine. If the default behavior isn't required, you must change the J2EE server configuration to use an existing JNDI server.

Using Sun Microsystems' J2EE Reference Implementation

Using JNDI with Sun Microsystems' J2EE Reference Implementation (RI) is straightforward. Ensure that

- The `J2EE_HOME` variable exists
- The `CLASSPATH` variable includes the `j2ee.jar` file from the `lib` directory of the J2EE home directory

Examples of how to do this both for Windows and for Unix are shown in this section.

J2EE RI for Windows

Under systems running Microsoft Windows NT or 2000, you can set the class path interactively with the following:

```
Set CLASSPATH=%J2EE_HOME%\lib\j2ee.jar;%CLASSPATH%
```

Typically, it is better to set the class path as a system-wide environment variable (via the `My Computer` properties dialog). A suitable value is as follows:

```
.;%J2EE_HOME%\lib\j2ee.jar
```

The class path can also include additional JAR files and directories for other Java components.

It is important to define the current directory (.) in the class path; otherwise, the Java compiler and runtime systems will not find the classes for the program being developed.

J2EE RI for Linux and Unix

Under Linux and Unix, set the class path with the following:

```
CLASSPATH=$J2EE_HOME/lib/j2ee.jar:$CLASSPATH
```

Starting the JNDI Server

Startup the J2EE RI server, as Day 2, "The J2EE Platform and Roles," described, and the JNDI server will start at the same time. You start the J2EE server by entering the following command from a command-line window:

```
j2ee –verbose
```

The J2EE server will run in that window until you close the window down or enter the following shutdown command from another command-line window:

```
j2ee -stop
```

Obtaining an Initial Context

The first step in using the JNDI name service is to get a context in which to add or find names. The context that represents the entire namespace is called the *Initial Context* and is represented by a class called `javax.naming.InitialContext` and is a sub-class of the `javax.naming.Context` class.

A `Context` object represents a context that you can use to look up objects or add new objects to the namespace. You can also interrogate the context to get a list of objects bound to that context.

The `javax.naming` package contains all the simple JNDI classes. Sub-packages within the `javax.naming` package provide additional JNDI functionality, such as directory-based features like attributes.

The following code creates an initial context using the default JNDI service information:

```
Context ctx = new InitialContext();
```

If something goes wrong when creating the initial context, a `NamingException` is thrown.

Initial Context Naming Exceptions

The runtime system reports errors in using JNDI as a subclass of `NamingException`. The exceptions most likely to occur for accessing the initial context are as follows:

```
javax.naming.CommunicationException: Can't find SerialContextProvider
```

This exception usually means the JNDI Server is not running, or possibly the JNDI properties for the server are incorrect (see the next section, “Defining the JNDI Service”).

```
javax.naming.NoInitialContextException:
➥Need to specify class name in environment or system property, or as an applet
➥parameter, or in an application resource file:  java.naming.factory.initial
```

This exception occurs when the `IntialContext` class does not have default properties for the JNDI Service Provider, and the JNDI server properties have not be configured explicitly (see the next section, “Defining the JNDI Service”).

```
javax.naming.NoInitialContextException: Cannot instantiate class: XXX
 [Root exception is java.lang.ClassNotFoundException: XXX]
```

This exception occurs when the class path defined for the JNDI program does not include the JNDI server classes (see the next section, “Defining the JNDI Service”).

```
javax.naming.ServiceUnavailableException:
➥Connection refused: no further information
[Root exception is java.net.ConnectException:
➥Connection refused: no further information]
```

This exception occurs when the JNDI properties for the program fail to match the JNDI Service Provider currently in use (see the next section, “Defining the JNDI Service”).

Defining the JNDI Service

During program development, it is reasonable to use a JNDI service running on the local machine that uses the default service provider supplied with the J2EE server. When you deploy the program, you must make use of the enterprise-wide naming service for your site. You will need to configure the program to use a specific naming server and not the default one provided with your test J2EE server.

The parameters that you usually need to define for the JNDI service are as follows:

- JNDI service classname
- Server’s DNS host name
- Socket port number

A particular server vendor’s implementation may require additional parameters.

There are several ways of defining the JNDI service properties for a program, but you only need to use one of them. You can either

- Add the properties to the JNDI properties file in the Java runtime home directory
- Provide an application resource file for the program

- Specify command-line parameters to be passed to an application
- Specify parameters to be passed into an applet
- Hard-code the parameters into the program

The last option is the weakest approach, because it restricts the program to working with one type of JNDI service provider on one specific host.

The first two options are the most suited to production environments. They both require that you distribute simple text configuration files with the program.

JNDI Properties Files

An application resource file called `jndi.properties` defines the JNDI service. The JNDI system automatically reads the application resource files from all components in the program's `CLASSPATH` and from `lib/jndi.properties` in the Java runtime home directory (this is the `jre` sub-directory of the Java JDK home directory).

The following example from Sun Microsystems' J2EE RI shows a typical `jndi.properties` file:

```
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
java.naming.provider.url=localhost:1099
java.naming.factory.url.pkgs=com.sun.enterprise.naming
```

Each entry in the property file defines a name value pair. The `InitialContext` object uses these properties to determine the JNDI service provider.

The J2EE server vendor usually supplies a sample `jndi.properties` file defining the properties that need to be configured with their server. You can find the J2EE RI `jndi.properties` file in the `lib/classes` directory in the J2EE RI installation directory.

Normally, any given JNDI service will require the following named properties:

```
java.naming.provider.url
```

This defines the DNS host name of the machine running the JNDI service and the service port number. This is the only property that the network administrator needs to customize. The property value is a machine name with an optional colon and port number. By default, JNDI uses port 1099, and most sites do not change this value. The default server is usually `localhost`.

Consider a host called `nameserver` in the Sams Publishing domain (`samspublishing.com`), the full URL including port number for a default JNDI server on this host would be as follows:

```
nameserver.samspublishing.com:1099
```

```
java.naming.factory.initial
```

You set this property to the classname (including the package) of the Initial Context Factory for the JNDI Service Provider. This value effectively determines which JNDI Service Provider you use. To use the default Naming Service supplied with the J2EE RI, you would specify this property as follows:

```
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
```

```
java.naming.factory.url.pkgs
```

This property defines prefix package names the `InitialContext` class uses for finding other classes JNDI requires. The J2EE RI uses the following value for this property:

```
java.naming.factory.url.pkgs=com.sun.enterprise.naming
```

More information on these and other JNDI properties can be found in the API documentation for the `Context` class and in the JNDI Tutorial from Sun Microsystems.

The simplest way to define the JNDI Service Provider is to configure every client's Java home directory to include the necessary JNDI properties. This approach suits an intranet where all machines are centrally managed.

Another approach is to include a suitable JNDI properties file with the client program and distribute everything as a JAR file (program class files and the `jndi.properties` file). This suits Web-based intranets or extranets, where applets are used or where you can distribute the client JAR file to users.

Application Properties

Using the `-D` option, you can supply the JNDI properties on the `java` command line of an application. This has the disadvantage of requiring long command lines that are hard to remember and easy to mistype. A way around this problem is for you to provide script files to run the application on the target platforms; typically, you will supply batch files for Windows and shell scripts for Linux and Unix.

The following is an example of a command line that defines the JNDI factory classes and server:

```
java
    ➥-D
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
    ➥-D java.naming.provider.url=localhost:1099 MyClass
```

Providing a `jndi.properties` file in the application JAR file is a cleaner solution than providing command-line parameters. However, using command-line parameters makes the JNDI properties more apparent when customizing the application for a local site. It is easy to overlook a `jndi.properties` file in a JAR file.

Applet Parameters

An applet can accept the JNDI properties as parameters, for example

```
<applet code="MyApplet.class" width="640" height="480">
<param name="java.naming.factory.initial"
    value= "com.sun.enterprise.naming.SerialInitContextFactory" >
<param name="java.naming.provider.url"
    "value= localhost:1099">
</applet>
```

Using parameters with the applet HTML file makes the JNDI properties more apparent when customizing the applet for a local site. A `jndi.properties` file in the jar file is easily overlooked.

Hard-Coded Properties

The least desirable way to specify the JNDI properties is via hard-coded values in the program. Hard coding the properties means including the JNDI classnames and the server name in the source code. This is undesirable because it means that should the network architecture change, you must edit, recompile, and redistribute the program. Obviously, you want to avoid this maintenance overhead if you can. The network architecture may change if the JNDI service moves to a different server or you install a new JNDI Service Provider.

The mechanism for defining the service in code is via a hash table of properties passed into the `InitialContext` constructor:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.enterprise.naming.SerialInitContextFactory");
env.put(Context. PROVIDER_URL,
    "localhost:1099");
Context ctx = new InitialContext(env);
```

Notice how the code uses symbolic constants from the `Context` class rather than using strings representing the properties (such as `"java.naming.factory.initial"`). This approach makes the code more portable should the property names change in future versions of Java or JNDI.

Binding JNDI Objects

After the initial JNDI context has been obtained, a program can look up existing objects and bind new objects to the context.

When working with EJBs, the main JNDI activity is to look up existing bound objects; the J2EE server does most of the binding of the objects automatically.

Because this section discusses the binding of objects, you can skip it if your primary purpose for using JNDI is to obtain EJB and other references within a J2EE application.

Binding Objects

Binding an object means adding a name to the JNDI service and associating that name with a Java object. The name and object are bound to a context. Listing 3.1 shows how a text message can be bound to the name `sams/book`.

LISTING 3.1 Full Text of `JNDIBind.java`

```
 1: import javax.naming.*;
 2: public class JNDIBind
 3: {
 4:     private final static String JNDI = "sams/book";
 5:
 6:     public static void main(String[] args) {
 7:         try {
 8:             Context ic = new InitialContext();
 9:             ic.bind(JNDI,"Teach Yourself J2EE in 21 Days");
10:             System.out.println("Bound "+JNDI);
11:         }
12:         catch (NamingException ex) {
13:              System.err.println(ex);
14:              System.exit(1);
15:         }
16:     }
17: }
```


The object to be bound must implement the `Serializable` interface so that the name server can store a copy of the object.

The `Context.bind()` method will fail with a `NameAlreadyBoundException` (which extends `NamingException`) if an object is already bound to that name. Another subclass of `NamingException` is thrown if there is some other form of error, such as an invalid name. Remember that different Service Providers may have different naming conventions.

Binding Problems

A Service Provider may not support binding of all types of objects. If the service cannot bind a particular object, it will throw an exception.

Using the default naming service for J2EE RI that uses a transient CORBA naming service, the class of the object must be in the `CLASSPATH` used by the J2EE RI JNDI server.

For now, this means using standard J2SE and J2EE classes or configuring the J2EE RI services to include your class files. The recommended approach is to edit the user

configuration file (`userconfig.sh` or `userconfig.bat`) in the `bin` directory of the J2EE RI home directory, and add the required class directories or JAR files to the `J2EE_CLASSPATH` variable defined in the configuration file.

An alternative is to use a Web Service to dynamically upload the required class files. Dynamic uploading of class files is dicussed in the "Loading Classes from a Code Base" section, later in this chapter.

Some Naming Services (such as LDAP) may use security features to ensure that only authorized programs can bind new objects. The `bind()` method can also fail if it violates any security features of the underlying naming service. The "Security" section of today's lesson covers this in more detail.

Name Persistence

A bound object normally remains in the namespace until it is unbound. If the bound name remains across server restarts, it is said to be persistent. Commercial servers, such as NDS, Active Directory, and LDAP, are persistent name servers and store the bound names and ancilliary information on disk (typically in a database).

The default naming service for Sun Microsystems' J2EE RI is a transient service; it reloads bound objects from configuration files in the SDK home directory whenever it is restarted. This naming service will not retain objects bound with the `Context.bind()` method across server restarts.

Rebinding Objects

You can use the `rebind()` method to solve the problem of `bind()` failing if a name is already bound. For example,

```
ic.rebind("sams/book","Teach Yourself J2EE in 21 Days");
```

The code unbinds any existing object bound to that name and binds the new object in its place.

Using `rebind()` is a good design technique when a programmer is sure the name will not be in use by another component. The alternative is to explicitly unbind the old name first if it is in use as discussed in the next section on "Unbinding Objects."

Unbinding Objects

You can remove an object from a namespace by using the `Context.unbind()` method. A program uses this method when it is closing down and needs to remove its advertised service because a bound name is not automatically unbound when the program shuts down.

Another common use for `unbind()` is to test if a name is already in use and unbind the old object before binding a new object. The advantage of using `unbind()` in preference to `rebind()` is that you can verify that the object to be unbound is at least of the same type as the new object to be bound.

```
String JNDI = "sams/book";
try {
    Object o = ic.lookup(JNDI);
    if (o instanceof String)
        ic.unbind (JNDI);
}
catch (NameNotFoundException ex) {}
ic.bind(JNDI,"Teach Yourself J2EE in 21 Days");
```

This example rebinds a string bound to the name `sams/book`, but will fail with a `NameAlreadyBoundException` if the name is bound to another class of object. This is a better design approach than that of using the `rebind()` method.

Renaming Objects

You can rename objects using `Context.rename()` by specifying the old name and then the new name as parameters. The new name must specify a name in the same context as the old name. An object must be bound to the old name, and the new name must not have a bound object; otherwise, a `NamingException` is thrown.

```
ic.rename("sams/book","sams/teachyourself");
```

JNDI Name Lookup

The most common use of JNDI is to look up objects that have been bound to a name. To do this, you require two items of information:

- The JNDI name
- The class of the bound object

With this information in hand, object lookup is the simple matter of using the `Context.lookup()` method to find the object and then to cast that object to the required class.

Listing 3.2 shows a simple program to look up the name `sams/book` that was bound by the program in Listing 3.1.

LISTING 3.2 Full Text of `JNDILookup.java`

```
1: import javax.naming.*;
2: public class JNDILookup
```

LISTING 3.2 Continued

```
 3: {
 4:     private final static String JNDI = "sams/book";
 5:     public static void main(String[] args) {
 6:         try {
 7:             Context ic = new InitialContext();
 8:             String name = (String)ic.lookup(JNDI);
 9:             System.out.println(JNDI+"="+name);
10:         }
11:         catch (NamingException ex) {
12:              System.err.println(ex);
13:              System.exit(1);
14:         }
15:         catch (ClassCastException ex) {
16:              System.err.println(ex);
17:              System.exit(1);
18:         }
19:     }
20: }
```

You can run the `JNDIBind` program in Listing 3.1 and then run this `JNDILookup` program to print out the value of the string bound against `sams/book`.

When casting an object that the `lookup()` method returns, that object's class must be in the client program's class path. If this is not the case, the program throws an exception.

Changing Contexts

The example name `sams/book` used in Listings 3.1 and 3.2 is an example of a *Composite Name*. If you need to look up many names in the same context of a composite name (names of the form `sams/...`), it is better to change to sub-context and look up the simple name within that context.

With this information in hand, the sub-context is a name entry just like any other name, and you look it up in just the same way. The retrieved object is another `Context` object. Listing 3.3 shows code that retrieves a name from a sub-context.

LISTING 3.3 Full Text of `JNDILookupSAMS.java`

```
1: import javax.naming.*;
2: public class JNDILookupSAMS
3: {
```

LISTING 3.3 Continued

```
 4:     public static void main(String[] args) {
 5:         try {
 6:             Context ic = new InitialContext();
 7:             Context ctx = (Context)ic.lookup("sams");
 8:             String name = (String)ctx.lookup("book");
 9:             System.out.println(name);
10:         }
11:         catch (NamingException ex) {
12:              System.err.println(ex);
13:              System.exit(1);
14:         }
15:         catch (ClassCastException ex) {
16:              System.err.println(ex);
17:              System.exit(1);
18:         }
19:     }
20: }
```


Narrowing RMI-IIOP Objects

There is only one additional twist to the lookup tale, and that is when dealing with RMI over IIOP objects.

The implementation of J2EE requires the use of RMI-IIOP to implement the remote interfaces to EJB components. Consequently, when a lookup is for an EJB name (more on this on Day 4, "Introduction to EJBs"), you cannot cast the returned object to the required class; instead, you must narrow it.

RMI-IIOP uses a portable remote object to encapsulate information about the real remote object. A portable remote object contains information about the real bound object in a portable format that can be interogated by the recipient to find the real remote object. The process of obtaining the real object from the portable remote object is called *narrowing*.

You use the `PortableRemoteObject.narrow()` method in the `javax.rmi` package to narrow a protable remote object to obtain the actual remote object. The narrow() method takes two parameters:

- The object to narrow
- A `java.lang.Class` object defining the real remote object's class

Listing 3.4 previews the discussion on Day 4 about EJB objects, but also serves to illustrate the use of the `narrow()` method.

LISTING 3.4 Narrowing an EJB Home Object

```
1: InitialContext ic = new InitialContext();
2: Object lookup = ic.lookup("java:comp/env/ejb/Agency");
3: AgencyHome home = (AgencyHome)
➥PortableRemoteObject.narrow(lookup, AgencyHome.class);
```

If your primary purpose for understanding JNDI is to enable the lookup and use of EJBs and other J2EE technologies (such as JDBC data sources and Message queues), you can skip the rest of this day's material and return to it at a later date.

Contexts

Contexts provide a hierarchical structure to JNDI names, and composite names group together related names. The initial context provides a top-level view of the namespace and any sub-contexts reflect the hierarchical composite name structure.

Listing Contexts

The namespace represents contexts as names, and you can look these up just like any other name. You can obtain a listing of the names in a context by using `Context.list()`. This method provides a list of name and class bindings as a `javax.naming.NamingEnumeration`, where each element in the enumeration is a `javax.naming.NameClassPair` object. Listing 3.5 shows a simple program to list the names and classes for the example `sams` sub context.

LISTING 3.5 Full Text of `JNDIListSAMS.java`

```
 1: import javax.naming.*;
 2: public class JNDIListSAMS
 3: {
 4:     public static void main(String[] args)
 5:     {
 6:         try {
 7:             Context ctx = new InitialContext();
 8:             NamingEnumeration list = ctx.list("sams");
 9:             while (list.hasMore()) {
10:                 NameClassPair item = (NameClassPair)list.next();
11:                 String cl = item.getClassName();
12:                 String name = item.getName();
13:                 System.out.println(cl+" - "+name);
14:             }
15:         }
16:         catch (NamingException ex) {
17:             System.out.println (ex);
18:             System.exit(1);
```

LISTING 3.5 Continued

```
        }
    }
}
```

You must run the `JNDIBind` program from Listing 3.1 before running this program; otherwise, the "`sams`" sub context will not exist. Running the program in Listing 3.5 will produce a single line of output:

```
java.lang.String - book
```

The parameter to the `list()` method defines the name of the context to list. If this is the empty string, the method lists the current context.

If the initial context of the J2EE RI namespace is listed, you must have the J2EE RI classes in your search path; otherwise, you will get an `org.omg.CORBA.BAD_PARAM` exception caused by another CORBA exception:

```
org.omg.CORBA.MARSHAL: Unable to read value from underlying bridge :
➥ Serializable readObject method failed internally
➥ minor code: 1398079699 completed: Maybe
```

Don't believe the `completed: Maybe` tagged on to the end of the error message. It didn't complete.

The easiest solution to this problem is to run the `setenv` script in the `bin` directory of J2EE RI. This script creates a variable `CPATH` that you can use as the `CLASSPATH` for running J2EE RI client programs.

Under Windows use

```
%J2EE_HOME%\bin\setenv
java -classpath .;%CPATH% JNDIList
```

Under Linux and Unix use

```
$J2EE_HOME/bin/setenv
java -classpath .:$CPATH JNDIList
```

The CD-ROM accompanying this book includes the `JNDIList` program, which is the same as the program in Listing 3.5 but without the parameter to the `list()` method.

The `list()` method returns the name and the bound object's classname, but not the object itself. It is a lightweight interface designed for browsing the namespace.

A second method, called `Context.listBindings()`, retrieves the object itself. The `listBindings()` method returns a `NamingEnumeration`, where each element is of type

`javax.naming.Binding`. Access methods in the `Binding` class support retrieval of the information of the bound object. Listing 3.6 shows a simple recursive tree-walking program that is a useful diagnostic tool for examining JNDI namespaces.

LISTING 3.6 Full Text of `JNDITree.java`

```
 1: import javax.naming.*;
 2: public class JNDITree
 3: {
 4:     public static void main(String[] args) {
 5:         Context ctx=null;
 6:         try {
 7:             ctx = new InitialContext();
 8:             listContext (ctx,"");
 9:         }
10:         catch (NamingException ex) {
11:             System.err.println (ex);
12:             System.exit(1);
13:         }
14:     }
15:
16:     private static void listContext (Context ctx, String indent) {
17:         try {
18:             NamingEnumeration list = ctx.listBindings("");
19:             while (list.hasMore()) {
20:                 Binding item = (Binding)list.next();
21:                 String className = item.getClassName();
22:                 String name = item.getName();
23:                 System.out.println(indent+className+" "+name);
24:                 Object o = item.getObject();
25:                 if (o instanceof javax.naming.Context)
26:                     listContext ((Context)o,indent+"  ");
27:             }
28:         }
29:         catch (NamingException ex) {
30:             System.err.println ("List error: "+ex);
31:         }
32:     }
33: }
```

Creating and Destroying Contexts

Binding a composite name will automatically create any intermediate sub-contexts required to bind the name. Binding the name `com/sams/publishing/book/j2ee in 21 days` creates the following sub-contexts if they don't already exist:

```
com
com/sams
com/sams/publishing
com/sams/publishing/book
```

You can explicitly create contexts with the `Context.createSubcontext()` method. The single method parameter is the name of the context. If this is a composite name, all intermediate contexts must already exist. The `createSubContext()` method will throw a `NameAlreadyBoundException` if the name already exists.

> Contrary to the API documentation, the J2EE RI naming service will automatically create any intermediate contexts.

Listing 3.7 shows a simple program for creating arbitrary contexts.

LISTING 3.7 Full Text of `JNDICreate.java`

```
 1: import javax.naming.*;
 2: public class JNDICreate
 3: {
 4:     public static void main(String[] args) {
 5:         try {
 6:             if (args.length != 1) {
 7:                 System.out.println ("Usage: JNDICreate context");
 8:                 System.exit(1);
 9:             }
10:             Context ic = new InitialContext();
11:             ic.createSubcontext(args[0]);
12:         }
13:         catch (NamingException ex) {
14:              System.err.println(ex);
15:              System.exit(1);
16:         }
17:     }
18: }
```

The `Context.destroySubcontext()` method can destroy contexts. Again, the single method parameter is the name of the context. The context does not have to be empty, because the method will remove from the namespace any bound names and sub-contexts with the destroyed context.

Listing 3.8 shows a simple program for deleting arbitrary contexts.

Use this program with caution, because destroying the wrong context will render your J2EE server unusable. If you are using the J2EE RI, restarting the J2EE server can rectify a mistake; this might not be the case with other servers.

LISTING 3.8 Full Text of `JNDIDestroy.java`

```
 1: import javax.naming.*;
 2: public class JNDIDestroy
 3: {
 4:     public static void main(String[] args) {
 5:         try {
 6:             if (args.length != 1) {
 7:                 System.out.println ("Usage: JNDIDestroy context");
 8:                 System.exit(1);
 9:             }
10:             Context ic = new InitialContext();
11:             ic.destroySubcontext(args[0]);
12:         }
13:         catch (NamingException ex) {
14:              System.err.println(ex);
15:              System.exit(1);
16:         }
17:     }
18: }
```

The `destroyContext()` method can throw a `NameNotFoundException` if the name doesn't exist and a `NotContextException` if the bound name is not a context.

More on JNDI Names

JNDI has to support different naming conventions for different Service Providers in the most transparent manner possible. Generally, programmers will specify JNDI names as strings, but a little understanding of how JNDI interprets bound names will help circumvent many simple problems that can occur when using names.

Special Characters

JNDI applies minimal interpretation to names specified as `String` objects. JNDI uses the forward slash character (`/`) as a name separator to provide a simple name hierarchy called a Composite Name. It is conventional for these composite names to be used to group related names (such as plumbers in the phone book). As an example, JDBC data sources take names of `jdbc/XXX` and EJBs the form `ejb/XXX`. While this is only a convention, it does help separate different sorts of named objects within the JNDI name space.

Composite and Compound Names

Composite names can span different naming systems. An LDAP name can combine with a file system name to get a composite name:

```
cn=Martin Bond, ou=Authors, o=SAMS, c=us/agency/agency.ldif
```

Here a filename (`agency/agency.ldif`) is appended to an LDAP name. How JNDI interprets this is up to the individual Service Provider.

Incidentally, JNDI calls structured names like the DNS and LDAP *compound names*. JNDI does not interpret compound names, but simply passes them through to the Service Provider.

Besides forward slash (`/`), JNDI also treats backslash (`\`), single quote (`'`), and double quote (`"`) characters as special. If a compound name or a component of a name contains any of these characters, they must be escaped using the backslash character (`\`).

If the underlying Service Provider uses the forward slash as a name separator (for example, the CORBA name service), there appears to be a conflict between JNDI and the Service Provider. In practice, there is unlikely to be a problem because JNDI recognizes two sorts of name separation—weak and strong. JNDI always passes the entire name to the Service provider. A strong name separation implementation (such as LDAP or DNS) simply processes the first part of the composite name and returns the remainder to the JNDI Naming Manager to pass on to other name services. A weak name separation implementation will simply process the entire composite name.The `COSNaming` server used in the J2EE RI uses weak separation, as does the `RMIRegistry` naming service.

For those programmers who need to do more that use names to look up and bind objects, JNDI provides several classes for manipulating and parsing composite and compound names. The JNDI name support classes in the `javax.naming` package are `Name`, `CompositeName`, and `CompoundName`.

URLs

In certain contexts, JNDI recognizes a URL (Uniform Resource Locator). The primary use of URLs is to identify the JNDI server usually through the `java.naming.provider.url` property, as shown in the following:

```
java.naming.provider.url=ldap://localhost:389
```

You can also specify a URL as a parameter to the `lookup()` and `bind()` methods in the `Context` class. For example,

```
Context ic = new InitialContext();
Object obj = ic.lookup("ldap://localhost:389/cn=Winston,dc=my-domain,dc=com");
```

This overrides the default context and forces JNDI to perform the lookup against the specified server. You need to take care with this approach, because the class path must contain the necessary Service Provider classes, and these must be able to process the request bind or lookup operation. In practice, this means that the URL must use the same Service Provider classes as the initial context.

Attributes

Attributes are a feature of a Directory service and are not available with simple name servers. Typically, you use attributes with an LDAP server. The J2EE RI JNDI server is a CORBA name server, and it does not support attributes.

An *attribute* is additional information stored with a name. Storing full name, address, phone number, and e-mail with a person's name is a common use of a directory service. NDS uses attributes to control access to shared network drives and to configure a user's login environment.

A directory service stores attributes as values against a keyword (LDAP calls them IDs). Directory services usually support searching for names (objects) that have certain attributes defined (or not defined). Searching often supports looking for names with certain attributes that have a specific value (often wildcard pattern matching is supported). A simple search of a personnel database under an LDAP server might be to find all names whose surname is Washington.

LDAP uses a schema system to control which attributes an object must define and those that it may define. Any attributes that you add or delete must not break the schema's requirements. LDAP servers may be able to disable schema checking, but this is usually a bad idea because the schema was created for a purpose.

If you want to see the capabilities of attributes, you must have access to a directory server. The rest of this section is based on using an LDAP Directory Server.

Overview of LDAP X.500 Names

LDAP names conform to the X.500 standard that requires a hierarchical namespace. A Distinguished Name (DN) unambiguously identifies each entry in the directory. The DN consists of the concatenation of the names from the root of the directory tree down to the specific entry.

X.500 focuses on the interoperability of different directory services rather than specifying how a directory service should define the DN. Consequently, different implementations of X.500 can each use a different syntax for representing object names (as shown earlier when we compared LDAP and Microsoft Active Directory names).

The official specification for the X.500 Directory Service is available from the International Telecommunications Union (ITU) Web site at `http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.500-199708-S`.

LDAP uses a comma-separated list of names with the names specified from the lowest entry up the tree to the higher entries.

Names consist of name and value pairs with the names typically being those in the following list. Each name has a short code and a full name; it is usual to only use the short code because the Distinguished Names are fairly long.

- `c` (`countryName`)—ISO two letter code for county such as `us`, `uk`, and so forth.
- `o` (`organizationName`)—Organization or company name, such as `samspublishing`
- `ou` (`organizationUnitName`)—Organizational unit, typically a division or department within an organization
- `l` (`localityName`)—Typically defines a location within an organizational unit
- `cn` (`commonName`)—Common name (sometimes called personal name), usually the name of the user or client
- `dc` (`domainComponent`)—A component part of a domain name (such as DNS names)
- `uid` (`userid`)—Typically represents a login name

An example LDAP DN looks like the following:

```
cn=Martin Bond, ou=Authors, o=SAMS, c=us
```

This will be a familiar structure if you work with digital certificates whose names conform to the X.509 standard.

Obtaining an LDAP Server

Using an LDAP Directory Service requires the JNDI properties to specify the JNDI Service provider from Sun Microsystems and to have an LDAP server running.

The J2EE RI does not include an LDAP server, so you will have to obtain one from elsewhere. Only certain operating systems provide LDAP servers. Windows NT, 2000, and XP users will have to purchase the enterprise (or server) editions of the operating system, which are typically significantly more expense than the usual desktop or professional editions. Sun Microsystems' Solaris 8 Operating Environment includes an LDAP server.

Linux and Unix users can download and install the OpenLDAP implementation, which is an open source server available free of charge for personal use. The Open LDAP server can be downloaded from `http://www.openldap.org/software/download/`.

Users of Microsoft Windows will have to make other arrangements as OpenLDAP is not available for the platform. If an Active Directory server is accessible on the network or you use the Enterprise (or Server) edition of the Operating System, this is a simple solution. Otherwise, Windows users are well advised to find a spare PC and install Linux on that and use OpenLDAP.

Interestingly, with today's low cost of hardware, it may be cheaper for the home user to purchase a second system to run Linux rather then purchase the additional license required to run the Enterprise (or Server) versions of Windows NT/2000/XP.

Using OpenLDAP

Given that many users experimenting with LDAP will probably use OpenLDAP on a Linux server, a brief digression on configuring Linux/LDAP for the programs in the rest of this section is useful. Other users must adapt the configuration for their own LDAP servers.

You only need to install Open LDAP if you want to evaluate the directory service features of JNDI and do not have access to a suitable directory service on your network. The following discussion of LDAP is not necessary for your understanding and use of J2EE.

The rest of this sub-section assumes that you have some knowledge of Linux and OpenLDAP (at the very least, that you have successfully installed OpenLDAP on a Unix server). The following OpenLDAP dicussion assumes that you have installed OpenLDAP in the default location of `/usr/local` (for example OpenLDAP v2.0.15 will be in the directory `/usr/local/openldap- 2.0.15`).

If you build and install OpenLDAP according to the supplied instructions, the process results in an empty LDAP directory namespace. The following steps will populate that namespace with a few sample entries. Make sure the `slapd` (LDAP) server is not running before making the following changes (if you have just installed OpenLDAP then `slapd` will not be running).

If you install OpenLDAP 2.0 with the recommended Berkeley database, it creates a default empty database with the DN suffix of `dc=my-domain,dc=com`.

To create a new DN suffix of `o=Agency,c=US`, you must add the following lines to the end of the `slapd` configuration file (`/usr/local/etc/openldap/slapd.conf`):

```
database        ldbm
suffix          "o=Agency,c=US"
directory       /usr/local/var/openldap-ldbm
rootdn          "cn=Manager,o=Agency,c=US"
rootpw          secret
index           objectclass     eq
index           cn,sn   pres,eq,sub,subany
```

Having defined the DN, you must add some sample data to the database. Listing 3.9 shows a configuration file for populating the database with sample data for use with the example programs shown in this section.

LISTING 3.9 Full Text of `agency.ldif`

```
 1: dn: o=Agency,c=us
 2: objectclass: top
 3: objectclass: organization
 4: o: Agency
 5: description: Job Agency
 6:
 7: dn: ou=Customers,o=Agency,c=us
 8: objectclass: top
 9: objectclass: organizationalUnit
10: ou: Customers
11:
12: dn: cn=All Customers,ou=Customers,o=Agency,c=us
13: objectclass: top
14: objectclass: groupofnames
15: member: cn=Winston,ou=Customers,o=Agency,c=us
16: member: cn=George,ou=Customers,o=Agency,c=us
17: cn: Customers
18:
19: dn: cn=Manager,o=Agency,c=us
20: objectclass: top
21: objectclass: person
22: cn: Manager
23: sn: Manager
24:
25: dn: cn=George,ou=Customers,o=Agency,c=us
26: objectclass: top
27: objectclass: person
28: cn: George
29: cn: George Washington
30: description: President
31: sn: Washington
32:
33: dn: cn=Abraham,ou=Customers,o=Agency,c=us
34: objectclass: top
35: objectclass: person
36: cn: Abraham
37: cn: Abraham Lincoln
38: description: President
39: sn: Lincoln
40:
41: dn: cn=Winston,ou=Customers,o=Agency,c=us
42: objectclass: top
43: objectclass: person
```

LISTING 3.9 Continued

```
44: cn: Winston
45: cn: Winston Churchill
46: description: Prime Minister
47: sn: Churchill
```

Copy this file to the Linux server and call it `agency.ldif`. Ensure that the `slapd` LDAP server is not running and, using the following `slapdadd` commandto install the sample names in the OpenLDAP configuration:

```
slapadd -b "o=Agency,c=US" -l agency.ldif
```

You can check the database configuration by using the `slapcat` command as follows:

```
slapcat | more
```

If you make a mistake or want to change the database configuration, you must delete the existing entries (`slapadd` will not replace an entry that already exists in the database). You can delete the existing OpenLDAP database by removing all of the files in the database directory specified in the `slapd.conf` configuration file. The following command will delete the default `ldbm` database used by `slapd`:

```
rm /usr/local/var/openldap-ldbm/*
```

You can now use `slapadd` to create the new database as shown previously.

You can start the `slapd` (OpenLDAP) server using the following command:

```
/usr/local/openldap-2.0.15/services/slapd/slapd -d 1
```

This will run the server in debug mode with diagnostic messages being displayed to the screen.

In the next section, you will test your LDAP server setup. You will find that the OpenLDAP database now has three entries for the `Customer` Organizational Unit:

```
cn=Abraham,ou=Customers,o=Agency,c=us
cn=George,ou=Customers,o=Agency,c=us
cn=Winston,ou=Customers,o=Agency,c=us
```

You must leave the `slapd` server running while you evaluate the examples shown in the rest of this section. When you want to stop the server, simply use Ctrl+C to interrupt the server or use the `kill` command to send the server a terminate signal.

Configuring JNDI to use LDAP

After an LDAP server is available for use, you must configure JNDI to use that server. This requires that you to obtain a JNDI Service Provider for LDAP (this isn't part of the LDAP server) and configure the JNDI properties accordingly.

JDK1.3 and J2EE RI 1.3 include an LDAP Service Provider from Sun Microsystems and all you have to do to use LDAP is to configure the JNDI properties to use the LDAP service. The simplest way to do this is to create an empty text file in the current directory called `jndi.properties` and add the following lines to this file.

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://localhost:389
```

If the LDAP server is not running on the current machine, replace the name `localhost` with the name or IP address of the actual LDAP server. Port number 389 is the default LDAP port number, and you can omit it if LDAP is running on the default port (or replace it by the actual port number if a non-standard port is being used).

Testing the LDAP Server

Before looking at attributes, it is worth checking that the LDAP server is up and running with the sample data. Verify that you can look up one of these sample names by using the simple `JNDILoopAny` script shown in Listing 3.10. The code reads the JNDI name from the command-line arguments.

LISTING 3.10 Full Text of `JNDILookupAny.java`

```
 1: import javax.naming.*;
 2: public class JNDILookupAny
 3: {
 4:
 5:     public static void main(String[] args) {
 6:         if (args.length != 1) {
 7:             System.err.println ("Usage: JNDILookupAny JNDIname");
 8:             System.exit(1);
 9:         }
10:         try {
11:             Context ic = new InitialContext();
12:             Object o = ic.lookup(args[0]);
13:             System.out.println(args[0]+"="+o);
14:         }
15:         catch (NamingException ex) {
16:              System.err.println(ex);
17:              System.exit(1);
18:         }
19:         catch (ClassCastException ex) {
20:              System.err.println(ex);
21:              System.exit(1);
22:         }
23:     }
24: }
```

Run the following command to check access to the LDAP server.

```
java JNDILookupAny "cn=Manager, o=Agency,c=us "
```

This will display an entry similar to the following:

```
ou=Customers,o=Agency,c=US=com.sun.jndi.ldap.LdapCtx@42719c
```

If the test doesn't work, you must check your local LDAP configuration. The most likely problem will relate to security. You may need to provide security credentials to the LDAP server. The "Security" section at the end of today's lesson briefly covers this.

Obtaining a Directory Context

Attributes are only supported by Directory Services and cannot be accessed through the ordinary `Context` object. Instead, you must use a `javax.naming.directory.DirContext class`. The `DirContext` is a sub-class of `Context`, and you can use it in place of a `Context` when dealing with a Directory Service where you require directory functionality (such as attributes). For example,

```
DirContext ic = new InitialDirContext();
```

The `javax.naming.directory` package contains the other attribute classes discussed next.

Reading Attributes

Attributes are read from the context just like a name is looked up from the context. The `DirContext.getAttributes()` method returns a `NamingEnumeration` that contains a collection of `Attribute` objects. Each `Attribute` has an ID (or key) and a list of values (an attribute can have more than one value for the same key). Listing 3.11 shows how all the attributes for an object can be listed.

LISTING 3.11 Full Text of `JNDIAttributes.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3: public class JNDIAttributes
 4: {
 5:     public static void main(String[] args) {
 6:         if (args.length != 1) {
 7:             System.err.println ("Usage: JNDIAttributes JNDIname");
 8:             System.exit(1);
 9:         }
10:         try {
11:             DirContext ctx = new InitialDirContext();
12:             Attributes attrs = ctx.getAttributes(args[0]);
```

LISTING 3.11 Continued

```
13:             NamingEnumeration ae = attrs.getAll();
14:             while (ae.hasMore()) {
15:                 Attribute attr = (Attribute)ae.next();
16:                 System.out.println("  attribute: " + attr.getID());
17:                 NamingEnumeration e = attr.getAll();
18:                 while (e.hasMore())
19:                     System.out.println("    value: " + e.next());
20:             }
21:             System.out.println("END of attributes for "+args[0]);
22:         }
23:         catch (NamingException ex) {
24:             System.out.println (ex);
25:             System.exit(1);
26:         }
27:     }
28: }
```


Running this program against the sample data produces the following result:

```
> java JNDIAttributes "cn=George,ou=Customers,o=Agency,c=us"
  attribute: description
    value: President
  attribute: objectClass
    value: person
  attribute: sn
    value: Washington
  attribute: cn
    value: George
    value: George Washington
END of attributes for cn=George,ou=Customers,o=Agency,c=us
```

A second form of the `getAttributes()` method allows you to provide an array of attribute names, and it only returns the values for those attributes. It is not an error to query an attribute that isn't defined; it simply doesn't return a value for that attribute. The following fragment shows how to find the `cn` and `sn` attributes for a name:

```
String[] IDs = {"sn", "cn"};
Attributes attrs = ctx.getAttributes("cn=George,ou=Customers,o=Agency,c=us",IDs);
```

Searching for Objects

A powerful and useful feature of attributes is the ability for you to search for names that have specific attributes or names that have attributes of a particular value.

You use the `Context.search()` method to search for names. There are several overloaded forms of this method, all of which require a DN to define the context in the name

tree where the search should begin. The simplest form of `search()` takes a second parameter that is an `Attributes` object that contains a list of attributes to find. Each attribute can be just the name or the name and a value for that attribute.

Listing 3.12 shows a simple program to find all names that have a surname (`sn`) defined and a `description` of `President`.

LISTING 3.12 Full Text of `JNDISearch.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3: public class JNDISearch
 4: {
 5:     private final static String JNDI = "ou=Customers,o=Agency,c=us";
 6:
 7:     public static void main(String[] args) {
 8:         try {
 9:             DirContext ctx = new InitialDirContext();
10:             // create case insensitive search attributes
11:             Attributes match = new BasicAttributes(true);
12:             match.put(new BasicAttribute("sn"));
13:             match.put(new BasicAttribute("description","president"));
14:             NamingEnumeration enum = ctx.search(JNDI, match);
15:             while (enum.hasMore()) {
16:                 SearchResult res = (SearchResult)enum.next();
17:                 System.out.println(res.getName()+","+JNDI);
18:             }
19:         }
20:         catch (NamingException ex) {
21:             System.out.println (ex);
22:             System.exit(1);
23:         }
24:     }
25: }
```

The `search()` method returns a `NamingEnumeration` containing objects of class `SearchResult` (a sub-class of `NameClassPair` discussed earlier). The `SearchResult` encapsulates information about the names found. The example program simply prints out the names (the names in the `SearchResult` object are relative to the context that was searched).

Running the program in Listing 3.12 will return the following values from the sample data:

```
cn=George,ou=Customers,o=Agency,c=us
cn=Abraham,ou=Customers,o=Agency,c=us
```

The `SearchResult` class also has a `getAttributes()` method that returns the attributes for the found name. The simple search shown in Listing 3.12 returns all of the name's attributes.

A second form of the `search()` method takes a third parameter that is an array of String objects specifying the attributes for the method to return. The following code fragment shows how to search and return just the surname and common name attributes:

```
NamingEnumeration enum = ctx.search(JNDI, match, new String[]{"sn","cn"});
```

Another form of the `search()` method takes a String parameter specifying a search filter. The filter uses a simple prefix notation for combining attributes and values. The JNDI API documentation and the JNDI Tutorial from Sun Microsystems provides full details of the search filter syntax. Listing 3.13 shows a search for names with a description or `President` or `Prime Minister`.

LISTING 3.13 Full Text of `JNDIFilter.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3: public class JNDIFilter
 4: {
 5:     private final static String JNDI = "ou=Customers,o=Agency,c=us";
 6:
 7:     public static void main(String[] args) {
 8:         try {
 9:             DirContext ctx = new InitialDirContext();
10:             SearchControls sc = new SearchControls();
11:             String filter =
➥"(|(description=President)(description=Prime Minister))";
12:             NamingEnumeration enum = ctx.search(JNDI, filter, sc);
13:             while (enum.hasMore()) {
14:                 SearchResult res = (SearchResult)enum.next();
15:                 System.out.println(res.getName()+","+JNDI);
16:             }
17:         }
18:         catch (NamingException ex) {
19:             System.out.println (ex);
20:             System.exit(1);
21:         }
22:     }
23: }
```

You can use the `javax.naming.directory.SearchControls` argument required by `search()` to

- Specify which attributes the method returns (the default is all attributes)
- Define the scope of the search, such as the depth of tree to search down to
- Limit the results to a maximum number of names
- Limit the amount of time for the search

Running the program in Listing 3.13 with the sample data produces the following output:

```
cn=George,ou=Customers,o=Agency,c=us
cn=abraham,ou=Customers,o=Agency,c=us
cn=Winston,ou=Customers,o=Agency,c=us
```

Manipulating Attributes

The `DirContext. ModifyAttributes()` method supports the addition, modification, and deletion of attributes for a name. To manipulate an attribute, the program must have write permission to entries in the LDAP name server. On a live system, the program must supply valid user credentials when obtaining the initial context (see the "Security" section later in this lesson). If you attempt to modify a name's attributes without the requisite permissions, a `javax.naming.NoPermissionException` is thrown.

If you are using the OpenLDAP server purely for evaluating JNDI, you can easily change the permissions so that all users have write permission. Find the `slapd` configuration file (by default, this is `/usr/local/etc/openladap/slapd.conf`) and replace the following line:

```
access to * by * read
```

with

```
access to * by * write
```

Stop and restart the `slapd` server for this change to take effect.

You can manipulate attributes in one of two ways. The first, and most functional, is to create an array of `javax.naming.directory.ModificationItem` objects. Each entry in the array specifies an attribute ID and an operation (one of `DirContext.REPLACE_ATTRIBUTE`, `DirContext.ADD_ATTRIBUTE`, and `DirContext.REMOVE_ATTRIBUTE`). To modify or add a new attribute, the `ModifyAttributes()` method requires an additional parameter for the value of the attribute.

Listing 3.14 shows how the entry for `Abraham` can update the `description` attribute and add a new `seeAlso` attribute.

LISTING 3.14 Full Text of `JNDIModify.java`

```
   1: import javax.naming.*;
   2: import javax.naming.directory.*;
   3: public class JNDIModify
   4: {
   5:     private final static String JNDI =
➥"cn=Abraham,ou=Customers,o=Agency,c=us";
```

LISTING 3.14 Continued

```
 6:
 7:     public static void main(String[] args) {
 8:         try {
 9:             DirContext ctx = new InitialDirContext();
10:             SearchControls sc = new SearchControls();
11:             ModificationItem[] mods = new ModificationItem[2];
12:             mods[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,
13:                 new BasicAttribute("description", "Assasinated
President"));
14:             mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE,
15:                 new BasicAttribute("seeAlso",
➥"cn=George,ou=Customers,o=Agency,c=us"));
16:             ctx.modifyAttributes(JNDI, mods);
17:         }
18:         catch (NamingException ex) {
19:             System.out.println (ex);
20:             System.exit(1);
21:         }
22:     }
23: }
```


After running this program, you can view the changes by using the `JNDIAttributes` program shown in listing 3.11.

```
> java JNDIAttributes "cn=abraham,ou=Customers,o=Agency,c=us"
  attribute: seeAlso
    value: cn=George,ou=Customers,o=Agency,c=us
  attribute: description
    value: Assasinated President
  attribute: objectClass
    value: top
    value: person
  attribute: sn
    value: Lincoln
  attribute: cn
    value: Abraham
    value: Abraham Lincoln
END of attributes for cn=abraham,ou=Customers,o=Agency,c=us
```

The second method for manipulating attributes is to define the operation and an `Attributes` list of `Attribute` objects to be manipulated (with values if appropriate). The next code fragment shows how to delete the `seeAlso` entry just added (which probably should have referred to John F. Kennedy and not George Washington).

```
Attributes attrs = new BasicAttributes("seeAlso",null);
ctx.modifyAttributes(JNDI, DirContext.REMOVE_ATTRIBUTE, attrs);
```

The next example shows how to change the `description` back to `President`.

```
Attributes attrs = new BasicAttributes("description","President");
ctx.modifyAttributes(JNDI, DirContext.REPLACE_ATTRIBUTE, attrs);
```

More on Objects

The early part of today's lesson covered binding objects into a JNDI namespace. To recap, a bound object must implement the `Serializable` interface, and the object's class file must be available to the JNDI server.

The obvious means of making an object's class file available to the JNDI server is to set the server's class path to include the necessary directory or JAR file. However, this isn't always convenient, and the JNDI specification recognizes this situation and supports dynamic loading of classes when using a directory service.

Loading Classes from a Code Base

Provided the JNDI Service Provider is a Directory Service and that service supports Internet RFC 2713, the JNDI server can obtain necessary class files dynamically from any HTTP server.

RFC2713 defines an interoperable way of storing Java objects in an LDAP server. By defining how Java objects are stored and retrieved, the JNDI Naming Manager in Sun Microsystems' LDAP Service Provider can retrieve Java objects from the directory server and recreate them on the client's system.

You must configure the LDAP server to support the Java Schema defined in RFC2713. If you are using the OpenLDAP server, as previously configured in the "Attributes" section, supporting the Java Schema is a relatively simple change to the configuration:

1. Stop the OpenLDAP `slapd` daemon.
2. Edit the `slapd` configuration file (`/usr/local/etc/openldap/slad.conf`). After the existing line starting with `include`, add the following line:

   ```
   include /usr/local/etc/openldap/schema/java.schema
   ```

3. Save the file and restart the OpenLDAP server.

Defining a Code Base

From a programming point of view, Java class files must be made available via a Web server. A `javaCodebase` attribute supplies the details of the Web server location when binding the object into the JNDI directory namespace.

The following example uses the Sun Microsystems' LDAP Service Provider and an LDAP server, as discussed in the "Attributes" section earlier in today's lesson.

This example also requires an HTTP server to be running. Starting up the J2EE RI server will also start an HTTP server on port 8000. The J2EE RI stores its HTTP pages in the `public_html` directory in the J2EE RI home directory.

Listing 3.15 shows a simple class representing a book.

LISTING 3.15 Full Text of `Book.java`

```
 1: import java.io.*;
 2: public class Book implements Serializable
 3: {
 4:     String title;
 5:     public Book(String title) {
 6:         this. title = title;
 7:     }
 8:     public String toString() {
 9:         return title;
10:     }
11: }
```


A Web server must make the `book.class` file available for download for the Sun Microsystems' LDAP Service provider to bind and look up `Book` objects. It is conventional to store downloadable class files in a separate sub-directory under the HTTP server's home directory. You normally call this sub-directory `classes`.

Use the following commands to copy the `book.class` file to the J2EE RI Web server. In both cases, you require appropriate permission to write to the J2EE RI home directory.

Under Windows

```
mkdir %J2EE_HOME%\public_html\classes
copy Book.class %J2EE_HOME%\public_html\classes
```

Under Linux and Unix

```
mkdir $J2EE_HOME/public_html/classes
cp Book.class $J2EE_HOME/public_html/classes
```

With the class files in place, a `Book` object can be bound to the LDAP namespace as Listing 3.16 shows. The code uses the `javaCodebase` attribute to specify the URL of the directory containing the Java class files and not the class file itself.

LISTING 3.16 Full Text of `JNDICodebase.java`

```
1: import javax.naming.*;
2: import javax.naming.directory.*;
3: public class JNDICodebase
4: {
```

LISTING 3.16 Continued

```
   5:     private final static String JNDI = "cn=book,o=Agency,c=us";
   6:     private final static String codeURL =
➥"http://localhost:8000/classes";
   7:
   8:     public static void main(String[] args) {
   9:         try {
  10:             DirContext ic = new InitialDirContext();
  11:             Book book = new Book("Teach Yourself J2EE in 21 Days");
  12:             Attributes attrs = new BasicAttributes();
  13:             attrs.put("javaCodebase", codeURL);
  14:             attrs.put("cn", "book");
  15:             ic.rebind(JNDI, book, attrs);
  16:             System.out.println("Bound "+JNDI);
  17:         }
  18:         catch (NamingException ex) {
  19:              System.err.println(ex);
  20:              System.exit(1);
  21:         }
  22:     }
  23: }
```

Note that the example uses the code base URL of `localhost:8000`, which is required because the J2EE web server uses a non-standard HTTP port (the standard HTTP port is 80).

The `http.port` entry in the `web.properties` file in the J2EE RI `config` directory defines the default port for the Web server.

With the name registered and the HTTP server running, the client can look up the bound object without having the `Book` class file in the search path. The Sun Microsystems' LDAP Service Provider automatically loads the class file. Listing 3.17 shows a simple client program.

LISTING 3.17 Full Text of `JNDILookupBook.java`

```
1: import javax.naming.*;
2: public class JNDILookupBook
3: {
4:     private final static String JNDI = "cn=book, o=Agency,c=us ";
5:
6:     public static void main(String[] args) {
7:         try {
```

LISTING 3.17 Continued

```
 8:             Context ic = new InitialContext();
 9:             Book book = (Book)ic.lookup(JNDI);
10:             System.out.println(JNDI+"="+book);
11:         }
12:         catch (NamingException ex) {
13:              System.err.println(ex);
14:              System.exit(1);
15:         }
16:         catch (ClassCastException ex) {
17:              System.err.println(ex);
18:              System.exit(1);
19:         }
20:     }
21: }
```


References

Sometimes, storing a serialized copy of an object in the Directory Service is inappropriate. Perhaps the object is too large or you must instantiate it dynamically because its construction depends on information that can vary from one client to another.

JNDI references provide a mechanism for storing an object by reference rather than by value. This mechanism only works if the underlying JNDI Service Provider supports `Referenceable` objects. The LDAP Service Provider from Sun Microsystems supports `Referenceable` objects.

Without going into too much detail, a reference to an object requires that a `Factory` class is available to build the object from the information the reference stores. From a design perspective, this requires two related classes:

- The `Object` class that must implement the `javax.naming.Referenceable` interface
- A `Factory` class that can create the required objects

The `Referenceable` interface requires an object to implement the `getReference()` method, which returns a `Reference` object. The `Reference` object defines the name of the class referred to and a `Factory` class that can be used to build the referenced object. Listing 3.18 shows a simple `Book` reference class.

LISTING 3.18 Full Text of `BookRef.java`

```
1: import java.io.*;
2: import javax.naming.*;
3: public class BookRef implements Referenceable
4: {
```

LISTING 3.18 Continued

```
 5:     String title;
 6:     public BookRef(String title) {
 7:         this.title = title;
 8:     }
 9:     public String toString() {
10:         return title;
11:     }
12:     public Reference getReference() throws NamingException {
13:         return new Reference(
14:             BookRef.class.getName(),
15:             new StringRefAddr("book", title),
16:             BookFactory.class.getName(),
17:             null);
18:     }
19: }
```

The second parameter to the `Reference` constructor uniquely defines the object. This requires a key to define the address type of the object (in this case, a `String` set to the value `book`) and a value for this specific object (in this case, the book's title). When the object is reconstructed, this address type and value will pass to the `Factory` object.

The `Factory` class must implement either `javax.naming.spi.ObjectFactory` or `javax.naming.spi.DirObjectFactory`, depending whether you use a Name Service or a Directory Service. Both classes require the `Factory` class to implement a `getObjectInstance()` method for creating reference objects. Listing 3.19 shows the `BookFactory` class used in the `BookRef` class shown in Listing 3.16.

LISTING 3.19 Full Text of `BookFactory.java`

```
 1: import javax.naming.*;
 2: import javax.naming.spi.*;
 3: import java.util.Hashtable;
 4: public class BookFactory implements ObjectFactory {
 5:     public Object getObjectInstance(Object obj, Name name,
 6:         Context ctx, Hashtable env) throws Exception {
 7:         if (obj instanceof Reference) {
 8:             Reference ref = (Reference)obj;
 9:             if (ref.getClassName().equals(BookRef.class.getName())) {
10:                 RefAddr addr = ref.get("book");
11:                 if (addr != null) {
12:                     return new BookRef((String)addr.getContent());
13:                 }
14:                }
15:         }
16:         return null;
17:     }
18: }
```

The factory `getObjectInstance()` method checks that it is passed a `Reference` object and then checks that the class of the reference is a `BookRef`. If both of these conditions are true, the factory uses the address type `book` to look up the value of the object and then uses this to create a new `BookRef` object.

As far as name binding and object lookup are concerned, the client is unaware of the use of references. Listings 3.20 and 3.21 show how your code can use the `BookRef` class.

LISTING 3.20 Full Text of `JNDIBindBookRef.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3: public class JNDIBindBookRef
 4: {
 5:     private final static String JNDI = " book";
 6:
 7:     public static void main(String[] args) {
 8:         try {
 9:             DirContext ic = new InitialDirContext();
10:             BookRef book = new BookRef("Teach Yourself J2EE in 21 Days");
11:             Attributes attrs = new BasicAttributes();
12:             attrs.put("cn", "book");
13:             ic.rebind(JNDI,book,attrs);
14:             System.out.println("Bound BookRef "+JNDI);
15:         }
16:         catch (NamingException ex) {
17:              System.err.println(ex);
18:              System.exit(1);
19:         }
20:     }
21: }
```

LISTING 3.21 Full Text of `JNDILookupBookRef.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3: public class JNDILookupBookRef
 4: {
 5:     private final static String JNDI = "book";
 6:
 7:     public static void main(String[] args) {
 8:         try {
 9:             DirContext ic = new InitialDirContext();
10:             BookRef name = (BookRef)ic.lookup(JNDI);
11:             System.out.println(JNDI+"="+name);
12:         }
13:         catch (NamingException ex) {
14:              System.err.println(ex);
```

LISTING 3.21 Continued

```
15:              System.exit(1);
16:         }
17:         catch (ClassCastException ex) {
18:              System.err.println(ex);
19:              System.exit(1);
20:         }
21:     }
22: }
```

What Else Can JNDI Do?

JNDI is a large subject, and some of the previous discussion has been quite brief (there is a lot more to attributes, searching, and references than has been shown). Today, the additional features, such as naming events and security, have been presented in a very superficial manner.

JNDI Events

JNDI supports an event model similar to the event listeners in the Java AWT and Swing classes. However, the underlying JNDI Service Provider must also provide support for the event model for a client to register event handlers.

The `javax.naming.event` package supports two types of JNDI event listener (both are sub-classes of `NamingListener`):

- `NamespaceChangeListener` reports on changes to the namespace objects that are added, removed, or renamed.
- `ObjectChangeListener` reports on changes to an object when its binding is replaced or attributes are added, removed, or replaced.

Both interfaces define appropriate methods that are called when changes occur in the JNDI namespace. A `javax.naming.event.NamingEvent` object is passed to the listener method to define:

- The type of event (for example, name added or object changed)
- The name binding before the event occurred
- The name binding after the event occurred

You use the `EventContext.addNamingListener()` method to register a `NamingListener` object against a context. Adding and removing a listener requires the context to implement the `EventContext` (or `EventDirContext` for Directory Services). The code to look up and register a `NamingListener` is similar to that shown in Listing 3.22 (please use the API documentation for further details):

LISTING 3.22 Sample Code to Add a `NamingListener`

```
1: Context ic = new InitialContext();
2: EventContext ctx = (EventContext) ic.lookup("sams");
3:
4: NamingListener listener = new MyNamingListener();
5:
6: ctx.addNamingListener("book", EventContext.ONELEVEL_SCOPE, listener);
```

Listing 3.22 registers a listener against the `sams` context and listens for events on the `book` name. Depending on the underlying Service Provider, the `book` name need not exist when you register the listener.

You can register a `NamingListener` to listen for several objects either by listening on a context (rather than an object) or by using attribute filters to specify the required objects.

JNDI event handling provides an effective means for monitoring changes to a namespace to maintain up-to-date information about the registered objects.

Security

JNDI security depends on the underlying Service Provider. Simple services, such as RMI and the CORBA name service (both of which the J2EE RI implementation supplies), do not support security. These services allow any client to perform any operation.

In a production environment, security is paramount to ensuring the integrity of the data in the JNDI server. Most live J2EE implementations will make use of LDAP to provide a naming service that supports security.

LDAP security is based on three categories:

- *Anonymous*—No security information is provided.
- *Simple*—The client provides a clear text name and password.
- *Simple Authentication and Security Layer (SASL)*—The client and server negotiate an authentication system based on a challenge and response protocol that conforms to RFC2222.

If the client does not supply any security information (as in all the examples shown today), the client is treated as an anonymous client.

The following JNDI properties provide security information:

- `java.naming.security.authentication` is set to a `String` to define the authentication mechanism used (one of `none`, `simple`, or the name of an SASL authentication system supported by the LDAP server).
- `java.naming.security.principal` is set to the fully qualified domain name of the client to authenticate.

- `java.naming.security.credentials` is a password or encrypted data (such as a digital certificate) that the implementation uses to authenticate the client.

If you do not define any of these properties, the implementation uses anonymous (`java.naming.security.authentication=none`) authentication.

You can use a JNDI properties file to supply client authentication information, but more usually you code the information within the client program. Usually, your application must obtain the client authentication dynamically.

If you use strong (not `simple` or `anonymous`) authentication, the `java.naming.security.authentication` value can consist of a space-separated list of authentication mechanisms. Depending on the LDAP service provider, JNDI can support the following authentication schemes:

- *External*—Allows JNDI to use any authentication system. The client must define a callback mechanism for JNDI to hook into the client's authentication mechanism.
- *GSSAPI (Kerberos v5)*—A well-known, token-based security mechanism.
- *Digest MD5*—Uses the Java Cryptography Extension (JCE) to support client authentication using the MD5 encryption algorithm, which has no known decryption technique.

Day 15, "Security," discuses the whole topic of J2EE and JNDI security.

Summary

JNDI provides a uniform API to an underlying naming or driectory service. A Naming Service provides a means of storing simple information against a name so the information can be retrieved using the name as a key. A Directory Service stores additional attribute information, as well as values against a name. Directory Services use attributes to categorize names so that powerful searching of the directory tree structure can be supported.

JNDI supports any naming service provided a Service Provider implementation is available for the service. Standard services supported by JNDI include the following:

- Lightweight Directory Access Protocol (LDAP)
- Novel Directory Services (NDS)
- CORBA
- Active Directory is supported via its LDAP interface

Using JNDI from within a Java program is a simple matter of creating a context and looking up names within that context. The `Context` class supports naming services, and the `DirContext` class supports directory services.

After a context has been defined, the `lookup()` method is used to retrieve the object stored against a name. The `bind()` and `rebind()` methods are used to add or changes bound objects, and the `unbind()` method is used to remove a bound object.

Within J2EE, JNDI is used to advertise components such as the following:

- EJBs
- Data sources (databases)
- JMS message queues and topics

Q&A

Q Why is a Name Service so important?

A Without JNDI, it would be a lot harder to provide services such as those implemented using J2EE objects like data sources, message queues, and EJBs. Each vendor would choose its own mechanism for defining how a client program should gain access to the J2EE objects. Some might do this by distributing configuration files, others by using TCP/IP broadcast network packets. Using a Name Service provides a consistent means of providing network services in a portable and platform-independent manner. Not only that, you can move an implementation of a service from one machine to another. In this instance, the server simply updates the Name Service entry to reflect its new location, and the whole process is transparent to the client.

Q Why is JNDI so large? Surely all I need to do is map a name onto a Java object?

A If all you want to do is support J2EE objects, JNDI could be as simple as a name to object mapping service. But Sun Microsystems designed JNDI to interoperate with established Directory Services, such as NDS, LDAP (Active Directory), and DNS. By providing Java programming support for these services, the designers of JNDI have ensured it will not be used as a proprietary product with J2EE servers, but as a general interface to fully -functional directory services. This design philosophy also provides programmers with a mechanism for developing interfaces to NDS and LDAP in Java rather than some other language, such as C++ or C#.

Exercise

You have been shown a simple program to display a JNDI namespace as a command line program (`JNDITree.java` in Listing 3.6). Today's exercise is to write a GUI version of this program using the Swing `JTree` class.If you already know Swing, you can use `JNDITree.java` as a guide for your program and go ahead and write your own JNDI browser.

If you do not know Swing, the exercise directory for Day 3 on the accompanying CD-ROM includes a template program called `JNDIBrowser.java` for you to enhance. The `JNDIBrowser` program handles all of the Swing initialization, all you have to do is get a list of the names in the JNDI namespace and create a new `javax.swing.tree.DefaultMutableTreeNode` representing the name and add this to the `JTree`. When you add a name that is also a context, you need to add all the names in this sub-context.

Comments have been added to the `JNDIBrowser.java` file to show you where to add your code.

Don't worry if this sounds complex—it isn't. You only have to write about 12 lines of code (most of which you can adapt from `JNDITree.java`).

Before you rush off and write your first piece of Java code for this book, remember that you need to set up your `CLASSPATH` to include the J2EE Reference Implementation classes using the `setenv` script, as discussed in today's lesson. You can run the supplied solution using the following commands.

Under Windows, use

```
%J2EE_HOME%\bin\setenv
java -classpath .;%CPATH% JNDIBrowser
```

Under Linux and Unix, use

```
$J2EE_HOME/bin/setenv
java -classpath .:$CPATH JNDIBrowser
```

If you complete this exercise or simply run the provided solution, you will see that the JNDI names are listed in the order they were added to the context. As a second exercise, change your program to display the names in alphabetical order. The solution called `JNDIBrowserSort.java` program shows how this can be achieved using the `java.util.TreeMap` class.

You will find these programs useful for browsing the JNDI namespace when you are developing J2EE applications.

WEEK 1

DAY 4

Introduction to EJBs

J2EE provides different types of components for different purposes. Today, you will start to look at one of the principal types of component in J2EE—Enterprise JavaBeans (EJBs).

The study of EJBs is continued on Day 5, “Session EJBs,”, Day 6, “Entity EJBs”, Day 7, “CMP and EJB QL”, Day 8, “Transactions and Persistence”, and Day 10, “Message-Driven Beans”. As you can see, there is a lot to learn about EJBs, so today serves as a first step on the road to all of this EJB knowledge.

Today, you will

- Examine the different types of EJB available
- Take a look at how EJBs are applied
- Explore the structure of one of the EJBs that forms part of the case study to see how the different parts fit together
- Deploy and use some of the EJBs from the case study
- Write a simple client for an EJB

First, you need to understand why you would use EJBs.

What Is an EJB?

In a typical J2EE application, Enterprise JavaBeans (EJBs) contain the application's business logic and live business data. Although it is possible to use standard Java objects to contain your business logic and business data, using EJBs addresses many of the issues you would find by using simple Java objects, such as scalability, lifecycle management, and state management.

Beans, Clients, Containers, and Servers

An EJB is essentially a managed component that is created, controlled, and destroyed by the J2EE container in which it lives. This control allows the container to control the number of EJBs currently in existence and the resources they are using, such as memory and database connections. Each container will maintain a pool of EJB instances that are ready to be assigned to a client. When a client no longer needs an EJB, the EJB instance will be returned to the pool and all of its resources will be released. At times of heavy load, even EJB instances that are still in use by clients will be returned to the pool so they can service other clients. When the original client makes another request of its EJB, the container will reconstitute the original EJB instance to service the request. This pooling and recycling of EJB instances means that a few EJB instances, and the resources they use, can be shared between many clients. This maximizes the scalability of the EJB-based application. The EJB lifecycle is discussed further on Days 5 and 6.

The client that uses the EJB instance does not need to know about all of this work by the container. As far as the client is concerned, it is talking to a remote component that supports defined business methods. How those methods are implemented and any magic performed by the container, such as just-in-time instantiation of that specific component instance, are entirely transparent to the client part of the application.

The EJB benefits from certain services provided by the container, such as automatic security, automatic transactions, lifecycle management, and so on. To do this, the EJB must conform to certain rules and implement an appropriate interface that allows the container to manage the component. The EJB is packaged with configuration information that indicates the component's requirements, such as transaction and security requirements. The container will then use this information to perform authentication and control transactions on behalf of the component—the component does not have to contain code to perform such tasks.

The primary purpose of the container is to control and provide services for the EJBs it contains. When it needs to use some underlying functionality, such as creating a transaction on behalf of a bean, it uses the facilities of the underlying EJB server. The EJB server is the base set of services on top of which the container runs. Different types of

EJB will run in different containers, but many different EJB containers can run on a single EJB server. EJB servers are generally delivered as part of a J2EE-compliant application server (examples include BEA WebLogic and IBM WebSphere). You will install and run the application server, which will provide the underlying services required of an EJB server and will host EJB containers.

The EJB Landscape

As you have seen, the J2EE Blueprints (`http://java.sun.com/blueprints/enterprise/index.html`) define a target architecture for a typical J2EE-based application. In this architecture, EJBs live in the middle tier and are used by other application components that live in the presentation tier. Although it is possible that both of these logical tiers will reside on the same computer, it is most likely that they will reside on different machines. This means that an EJB will usually have to be made available to remote clients.

To offer services to remote clients, EJBs will export their services as RMI remote interfaces. RMI allows you to define distributed interfaces in Java. There are certain caveats on doing this, not only at the implementation level (such as declaring that `RemoteExceptions` may be thrown when calling a method on an EJB) but also at the design level. Designing remote interfaces is a skill in itself, which will be explored as you progress through topics in this book, such as EJBs and J2EE Patterns.

Because they must use an RMI-based interface to access the functionality of the EJB, the clients of an EJB must have some programming functionality. This means that they are typically either "thick" clients that provide a GUI interface or Web-server components that deliver HTML interfaces to "thin" clients. The different types of client are explored in more detail shortly.

In the other direction, EJBs themselves will make use of data sources, such as databases and mainframe systems, to perform the required business logic. Access to such data and services can be through a JDBC database connection, a J2EE Connector, another EJB, or a dedicated server or class of some form.

Discovering EJBs

While it is quite easy to draw pictures of a 3-tier system containing boxes labelled "EJB," it is important to identify what application functionality should go into an EJB.

At the start of application development, regardless of the precise development process used (Rational Unified Process (RUP), eXtreme Programming (XP), and so on), there is generally some analysis that delivers a Unified Modelling Language (UML) domain model (this identifies the main elements of the business problem to be solved). This can

then form the basis of a solution model where the business concepts are mapped into appropriate design-level artefacts, such as components. This is where EJBs come into the design.

The UML model will consist of a set of classes and packages that represent single or grouped business concepts. A class or package can be implemented as an EJB. Generally, only larger individual classes will become EJBs in themselves, because EJBs are intended to be fairly coarse-grained components that incorporate a reasonably large amount of functionality and/or data.

There are generally two types of functionality discovered during analysis—data manipulation and business process flow. The application model will usually contain data-based classes such as `Customer` or `Product`. These classes will be manipulated by other classes or roles that represent business processes, such as `Purchaser` or `CustomerManager`. There are different types of EJB that can be applied to these different requirements.

Types of EJB

There are three different types of EJB that are suited to different purposes:

- *Session EJB*—A Session EJB is useful for mapping business process flow (or equivalent application concepts). There are two sub-types of Session EJB — stateless and stateful— that are discussed in more detail on Day 5. Session EJBs commonly represent "pure" functionality that is created as it is needed.
- *Entity EJB*—An Entity EJB maps a combination of data (or equivalent application concept) and associated functionality. Entity EJBs are usually based on an underlying data store and will be created based on that data within it.
- *Message-driven EJB*—A Message-driven EJB is very similar in concept to a Session EJB, but is only activated when an asynchronous message arrives.

As an application designer, you should choose the most appropriate type of EJB based on the task to be accomplished.

Common Uses of EJBs

So, given all of this, where would you commonly encounter EJBs and in what roles? Well, the following are some examples:

- In a Web-centric application, the EJBs will provide the business logic that sits behind the Web-oriented components, such as servlets and JSPs. If a Web-oriented application requires a high level of scalability or maintainability, use of EJBs can help to deliver this.

- Thick client applications, such as Swing applications, will use EJBs in a similar way to Web-centric applications. To share business logic in a natural way between different types of client applications, EJBs can be used to house that business logic.
- Business-to-business (B2B) e-commerce applications can also take advantage of EJBs. Because B2B e-commerce frequently revolves around the integration of business processes, EJBs provide an ideal place to house the business process logic. They can also provide a link between the Web technologies frequently used to deliver B2B and the business systems behind.
- Enterprise Application Integration (EAI) applications can incorporate EJBs to house processing and mapping between different applications. Again, this is an encapsulation of the business logic that is needed when transferring data between applications (in this case, in-house applications).

These are all high-level views on how EJBs are applied. There are various other EJB-specific patterns and idioms that can be applied when implementing EJB-based solutions. These are discussed more on Day 18, "Patterns."

Given this context, common types of EJB client include the following:

- A servlet or JSP that provides an HTML-based interface for a browser client
- Another EJB that can delegate certain of its own tasks or can work in combination with other EJBs to achieve its own goals
- A Java/Swing application that provides a front-end for the business processes encapsulated in the EJB
- A CORBA application that takes advantage of the EJB's business logic
- An applet that takes advantage of the business logic in a remote EJB so that this business logic does not need to be downloaded to the client

These are common ways that EJBs are applied. What benefits does the use of EJBs give to you as a developer?

Why Use EJBs?

Despite the recommendations of the J2EE Blueprints, the use of EJBs is not mandatory. You can build very successful applications using servlets, JSPs or standalone Java applications.

As a general rule of thumb, if an application is small in scope and is not required to be highly scalable, you can use J2EE components, such as servlets, together with direct JDBC connectivity to build it. However, as the application complexity grows or the number of concurrent users increases, the use of EJBs makes it much easier to partition and scale the application. In this case, using EJBs gives you some significant advantages.

Hiding Complexity

Early middleware environments, such as "raw" CORBA, require the application developer to write a lot of code that interacts with the CORBA environment and facilitates the connectivity and registration process. Such code can be likened to the plumbing that pipes water around a house. It needs to be there but, as the user of a sink or shower, you do not want to be intimately involved with it. In J2EE application terms, business developers want to write business code, not "plumbing" code. The EJB model tries to reduce such interaction to a minimum by using the following mechanisms:

- Each bean conforms to a defined lifecycle and set of rules. This provides a distinct boundary between system code and application code.
- Declarative attributes allow a developer to specify, say, the transactional behavior of the component without having to write code to control such functionality.
- The deployment information provided with the deployable J2EE application provides information about the relationships between multiple EJBs and also defines the resources required by an EJB.

Separation of Business Logic from UI and Data Access

One of the key facets of applying EJBs is that they allow business functionality to be developed and then deployed independently of the presentational layer. You might, for example, create an application with a user interface built using Java's Swing API. This application might then provide access to some business functionality for the employees working on the company's internal network. If the underlying business functionality is implemented using EJBs, a different user interface could take its place without having to redevelop the entire application. A Web-based interface that used servlets would make the application available from the Internet without having to change a single line of code in the business functionality. Figure 4.1 is a UML component diagram that shows this. (More information on UML can be found in Appendix A, "An Introduction to UML," on the accompanying CD-ROM.)

It can sometimes be difficult to distinguish between the functionality that an application provides and the user interface that is used to invoke that functionality. This is not unexpected because many common applications—such as a word-processor—are single-tier; the presentational logic and the business functionality are a single entity. On the other hand, consider programming a video recorder. Most modern video recorders can be programmed either directly on the console or through a remote control unit. Either user interface will accomplish the task of recording your favorite TV show, but there is only a single "application."

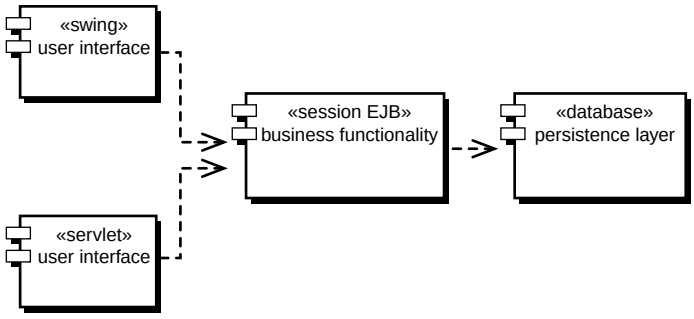


FIGURE 4.1
An application implemented using EJBs can have more than one user interface.

Consider another example. In most supermarkets, a cashier is responsible for scanning the items in your shopping cart and then requesting a payment for the total. However, some supermarkets also offer a trust system, whereby the customer scans the items with a mobile scanner as they place the item into the shopping cart. To pay for the goods in the shopping cart, the customer simply swipes his or her own card, and then leaves with the goods. Again, there is a single application (to purchase shopping items) but two different interfaces—the cashier's till and the customer's mobile scanner.

To implement a distributed application using EJBs, make sure you have distinguished between the user interface and the underlying business function. The EJB itself is concerned only with the latter of these.

Container Services

The container provides various services for the EJB to relieve the developer from having to implement such services, namely

- *Distribution via proxies*—The container will generate a client-side stub and server-side skeleton for the EJB. The stub and skeleton will use RMI over IIOP to communicate.
- *Lifecycle management*—Bean initialization, state management, and destruction is driven by the container, all the developer must do is implement the appropriate methods.
- *Naming and registration*—The EJB container and server will provide the EJB with access to naming services. These services are used by local and remote clients to look up the EJB and by the EJB itself to look up resources it may need.
- *Transaction management*—Declarative transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.
- *Security and access control*—Again, declarative security provides a means for the developer to easily delegate the enforcement of security to the container.

- *Persistence (if you want)*—Using the Entity EJB's container-managed persistence mechanism, state can be saved and restored without having to write a single line of code.

All of these container services are covered in more detail as the book progresses.

Now that you know why you would want to use an EJB and how to apply it, you can examine the inner workings of an EJB to understand how all the parts fit together.

What's in an EJB?

So far, you have been presented with a "black box" view of an EJB; it provides business functionality via an RMI remote interface, and it cooperates with its container to perform its duties. To understand, use, and ultimately write EJBs, you will need to know more in concrete terms about the Java programming artefacts that make up an EJB. In other words, what's in one?

The Business Interface

The primary purpose of an EJB is to deliver business or application logic. To this end, the bean developer will define or derive the business operations required of the bean and will formalize them in an RMI remote interface. This is referred to as the bean's business or remote interface as opposed to the home interface you will look at in a moment.

You may see references to local EJB interfaces and wonder how these relate to the current discussion. Don't worry about local interfaces for the moment; they are covered on Day 6 when you examine entity EJBs.

The actual methods defined on the remote interface will depend on the purpose of the bean, but there are certain general rules concerning the interface:

- As with any RMI-based interface, each method must be declared as throwing `java.rmi.RemoteException` in addition to any business-oriented exceptions. This allows the RMI subsystem to signal network-related errors to the client.
- RMI rules also apply to parameters and return values, so any types used must either be primitive, `Serializable`, or `Remote`.
- The interface must declare that it extends the `javax.ejb.EJBObject` interface. This provides a handful of basic methods that you will encounter as you progress.

Failure to conform to the rules about extending `javax.ejb.EJBObject` and throwing `RemoteException` will cause the interface to be rejected by tools that manipulate EJBs. Additionally, if you use parameter or return types that do not conform to the rules, your bean will compile and even deploy, but will fail with runtime errors.

The issue regarding object parameters and return values is worth considering for a moment. When you pass a parameter into a local method call, a reference to the original object is provided to be used within the method. Any changes to the state of the object are seen by all users of that object because they are sharing the same object. Also, there is no need to create a copy of the object—only a reference is passed.

On the other hand, when using RMI remote methods, objects that are serializable (implement the `Serializable` interface) are passed by value, whereas objects that are remote (that is, EJBs) are passed by reference. *Pass by value* means that a copy of the object is sent. This has several implications. First, users of a serializable object passed across a remote interface will no longer share the same object. Also, there may now be some performance costs associated with invoking a method through a bean's remote interface. Not only is there the cost of the network call, but also there is the cost of making a copy of the object so that it can be sent across the network. Most of the time, it will be serializable objects that are passed.

You can see an example of an EJB remote interface in Listing 4.1—in this case, the one for the `Agency` EJB used in the case study.

LISTING 4.1 Remote Interface for the `Agency` EJB

```
package agency;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface Agency extends EJBObject
{
    String getAgencyName() throws RemoteException;

    Collection findAllApplicants()
      throws RemoteException;
    void createApplicant(String login, String name, String email)
      throws RemoteException, DuplicateException, CreateException;
    void deleteApplicant (String login)
```

LISTING 4.1 Continued

```
        throws RemoteException, NotFoundException;

    Collection findAllCustomers() throws RemoteException;
    void createCustomer(String login, String name, String email)
      throws RemoteException, DuplicateException, CreateException;
    void deleteCustomer (String login)
      throws RemoteException, NotFoundException;

    Collection getLocations()
      throws RemoteException;
    void addLocation(String name)
      throws RemoteException, DuplicateException;
    void removeLocation(String code)
      throws RemoteException, NotFoundException;

    Collection getSkills()
      throws RemoteException;
    void addSkill(String name)
      throws RemoteException, DuplicateException;
    void removeSkill(String name)
      throws RemoteException, NotFoundException;

    List select(String table)
      throws RemoteException;
}
```

The interface lives in a package called `agency`, which will be common to all the classes that comprise the EJB. The definition imports `java.rmi.*` and `javax.ejb.*` for `RemoteException` and `EJBObject`, respectively. The rest of the interface is much as you would expect from any remote Java interface—in this case, passing `Strings` and returning serializable `Collections`.

Notice that all the methods must be declared as throwing `RemoteException`. This means that the client will have to handle potential exceptions that may arise from the underlying distribution mechanism. However, your application will probably want to employ exceptions itself to indicate application-level errors. These exceptions should be declared as part of the remote interface, as shown by the use of `NotFoundException` and `DuplicateException` in the `Agency` interface.

The Business Logic

After an interface is defined, there is the none-too-trivial task of implementing the business logic behind it. The business logic for an EJB will live in a class referred to as the bean. The bean consists of two parts:

- The business logic itself, including implementations of the methods defined in the remote interface
- A set of methods that allow the container to manage the bean's lifecycle.

Although the bean itself must contain these elements, note that it is possible, indeed common, for non-trivial beans to delegate some or all of their business functionality to other, helper, classes.

Drilling down into these areas reveals more about the structure of an EJB.

Implementing the Business Interface

The first thing to note is that the bean itself does **not** implement the remote interface previously defined. This may seem slightly bizarre at first sight, because the equivalent RMI server would *have to* implement the associated remote interface. However, there is a very good reason for this.

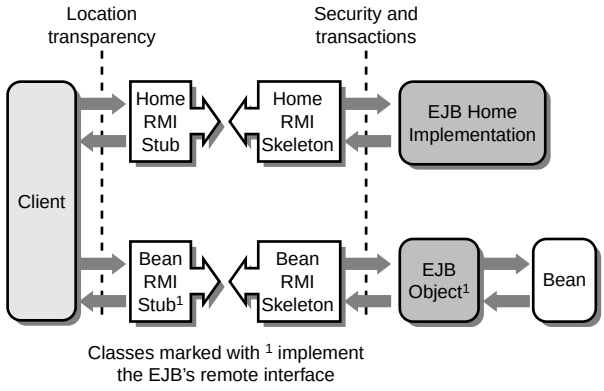
As you will see later, it is possible to ask the container to apply services, such as access control, on behalf of the EJB simply by setting attributes in the EJB configuration information. To do this, the container must have some way of intercepting the method call from the client. When it receives such a method call, the container can then decide if any extra services need to be applied before forwarding the method call on to the bean itself. Sticking with the security example, the container would examine security information configured for the EJB before deciding whether to forward the method call to the bean or to reject it. The details about access control are covered on Day 15, "Security," but you can see that it is necessary to interpose between the client and the bean to "automagically" deliver such services.

The interception is performed by a server-side object called the EJBObject (not to be confused with the interface of the same name). The EJBObject acts as a server-side proxy for the bean itself, and it is the EJBObject that actually implements the EJB's remote interface. Figure 4.2 shows the relationship between the client, the bean, and the EJBObject.

As shown in Figure 4.2, the client calls the business methods on the EJBObject implementation. The EJBObject applies the required extra services and then forwards the method calls on to the bean itself. The EJBObject is separate from the RMI stub and skeleton that provide the remote procedure call capability.

FIGURE 4.2

The EJBObject acts as a server-side proxy for the bean itself.



So, your bean must implement the business methods defined in the remote interface. The container uses the method signatures defined in the interface, together with the Java reflection API, to find the appropriate methods on the bean, so you must ensure that you use the correct method signatures. Despite this, the bean should not implement the remote interface itself (the reasons for this are discussed later). However, if you are using a developer tool that supports the creation of EJBs, it will generally generate empty methods for you to populate. Listing 4.2 contains the outlines of the business methods in the example `AgencyBean`.

LISTING 4.2 Business Method Implementation Signatures for the `AgencyBean`

```
package agency;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
// Remaining imports removed for clarity

public class AgencyBean implements SessionBean
{
    public String getAgencyName() {
      // Code removed for clarity
    }

    public Collection findAllApplicants() {
      // Code removed for clarity
    }

    public void createApplicant(String login, String name, String email)
      throws DuplicateException, CreateException {
      // Code removed for clarity
    }
```

LISTING 4.2 Continued

```
    public void deleteApplicant (String login)
      throws NotFoundException {
      // Code removed for clarity
    }

    public Collection findAllCustomers() {
      // Code removed for clarity
    }

    public void createCustomer(String login, String name, String email)
      throws DuplicateException, CreateException {
      // Code removed for clarity
    }

    public void deleteCustomer (String login) throws NotFoundException {
      // Code removed for clarity
    }

    public Collection getLocations() {
      // Code removed for clarity
    }

    public void addLocation(String name) throws DuplicateException {
      // Code removed for clarity
    }

    public void removeLocation(String code) throws NotFoundException {
      // Code removed for clarity
    }

    public Collection getSkills() {
      // Code removed for clarity
    }

    public void addSkill (String name) throws DuplicateException {
      // Code removed for clarity
    }

    public void removeSkill (String name) throws NotFoundException {
      // Code removed for clarity
    }

    public List select(String table) {
      // Code removed for clarity
    }

      // Remaining methods removed for clarity
}
```

The detail of the method implementations have been removed for clarity, because the main area of interest here is how the method signatures match up with those on the remote interface. The contents of the methods are largely the creation and dispatch of JDBC statements and handling the results from the queries.

Note that the bean does not implement the `Agency` interface. You can also see that various of the methods, such as `addSkill()`, declare that they throw an application-specific exception—in this case, `DuplicateException`.

Note that your bean methods will only throw business exceptions or standard Java exceptions. They should not throw `java.rmi.RemoteException`, because such exceptions should only be generated by the RMI subsystem.

Providing Lifecycle Hooks

Remember that the intention of the EJB environment is that you will spend most of your time writing business logic rather than network and database "plumbing." Beyond writing the business logic, the only additional thing the bean writer needs to do is to provide lifecycle "hooks" that allow the container to manage the bean.

Each of the different types of EJB discussed earlier has a slightly different lifecycle, but the common parts are as follows:

- Bean creation and initialization
- Bean destruction and removal
- The saving and restoring of the bean's internal state (if applicable)

The details associated with each type of bean lifecycle will be discussed as they are covered. For now, all you need to know is that

- An EJB will implement one or more lifecycle interfaces depending on its type. The interfaces (`SessionBean`, `EntityBean`, `MessageDrivenBean`, and `SessionSynchronization`) are defined in the `javax.ejb` package.
- The lifecycle methods will generally begin with `ejb` so that they can be easily distinguished from the business methods around them, for example, `ejbCreate()`.

Listing 4.3 contains the lifecycle methods in the example `AgencyBean`.

LISTING 4.3 Lifecycle Methods on the AgencyBean

```
package agency;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
// Remaining imports removed for clarity

public class AgencyBean implements SessionBean
{
    private DataSource dataSource;
    private String name = "";

    private void error (String msg, Exception ex) {
        String s = "AgencyBean: " + msg + "\n" + ex;
        System.out.println(s);
        throw new EJBException(s);
    }

    public void ejbCreate () throws CreateException {
        try {
            InitialContext ic = new InitialContext();
            dataSource = (DataSource)ic.lookup("java:comp/env/jdbc/Agency");
            name = (String)ic.lookup("java:comp/env/AgencyName");
        }
        catch (NamingException ex) {
            error("Error connecting to java:comp/env/Agency:", ex);
        }
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
        dataSource = null;
    }

    private SessionContext ctx;

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

      // Remaining methods removed for clarity
}
```

As you can see, the example `AgencyBean` implements the `SessionBean` interface. This means that it must implement the `ejbCreate()`, `ejbRemove()`, `ejbActivate()`, `ejbPassivate()`, and `setSessionContext()` methods. The `ejbCreate()` method takes on the role of constructor in that most of the bean initialization will take place in there. The context passed in `setSessionContext()` provides a way for the bean to communicate with the container.

This concludes the examination of the bean internals for the time being. You will discover more as you learn about the specific types of EJB later.

Factory Information

For an EJB to be used by a client, the client must create a new instance or discover an existing one. Finding and gaining access to the services of a traditional remote server is relatively simple. Such a server will tend to start when the machine boots, reside in a well-know location, and carry on running until the machine shuts down. However, EJBs are far more dynamic than that. It is the ability to dynamically create and reuse beans that provides the scalability inherent in the EJB model.

To facilitate the creation and discovery of EJBs, each type of EJB provides a home interface. The bean developer will provide an EJB home interface that acts as a factory for that particular EJB. A home interface will extend the `javax.ejb.EJBHome` interface and will contain the necessary methods identified by the bean developer that allow a client to create, find, or remove EJBs.

There are two ways for a client to get hold of the EJB itself, depending on the type of EJB (Session, Entity, or Message-driven) and the way it is intended to be used. The EJB Home interface can contain one or more `create()` methods to create a new instance of an EJB. So, for example, you will create a new instance of a Session bean before using it. On the other hand, when you interact with Entity EJBs, you will frequently find existing EJBs using one or more `findXXX()` methods. The home interface may or may not allow you to remove the bean, depending on bean type and usage.

Listing 4.4 shows the home interface for the example `Agency` EJB.

LISTING 4.4 Home Interface for the `Agency` Bean

```
package agency;

import java.rmi.*;
import javax.ejb.*;

public interface AgencyHome extends EJBHome
{
```

LISTING 4.4 Continued

```
    Agency create () throws RemoteException, CreateException;
}
```

Because the `Agency` EJB is just a simple wrapper around some JDBC-based functionality and does not maintain any business state, all that is required is a simple creation method—`create()`. This maps onto the `ejbCreate()` seen in Listing 4.3. The client will call `create()` to create an instance of the `Agency` bean.

The code underlying the home interface will work with the container to create, populate, and destroy EJBs as requested by the client. The effects of the method calls will vary depending on the type of EJB being manipulated. As a result, a request to remove a Session EJB will just result in the EJB being thrown away, while the same request on an Entity EJB may cause underlying data to be removed. The types and effects of different home interface methods are discussed in more detail on subsequent days.

Bean Metadata

The final piece of the EJB jigsaw lies in the provision of configuration information, or metadata, for the EJB. This provides a way of communicating the EJB's requirements and structure to the container. If an EJB is to be successfully deployed, the container will have to be provided with extra information, including

- An identifier or name for the EJB that can be used to look it up.
- The bean type (Session, Entity, or Message-driven).
- Which class is the EJB's remote interface. This interface will typically just be named according to the EJB's functionality, for example, `Agency` or `BankTeller`.
- Which class is the EJB's home interface. The name for an EJB's home interface will typically be derived from its remote interface name. So, for example, the `Agency` EJB has a home interface called `AgencyHome`. However, because this is a convention rather than being mandatory, the metadata explicitly indicates the name of the home interface.
- Which class is the bean itself. Again, the name for the bean will typically be derived from the associated remote interface name. So, for example, the `Agency` bean is called `AgencyBean`. However, because this is a convention rather than being mandatory, the metadata explicitly indicates the name of the bean.
- Any name/value pairs to be provided as part of the bean's environment.
- Information about any external resources required by the EJB, such as database connections or other EJBs.

All of this essential information is bundled into a deployment descriptor that accompanies the EJB classes. As you might expect, given its recent rise as the most ubiquitous way to define data, the deployment descriptor is defined as an XML document. The deployment descriptor is discussed in more detail soon when examining the packaging of an EJB.

In addition to the essential information, the deployment descriptor can also carry other metadata that you will encounter as you progress:

- Declarative attributes for security and transactions
- Structural information bean relationships and dependencies
- Persistence mapping (if applicable)

You are now nearing the conclusion of this whistle-stop tour of the structure of an EJB. After you have examined how an EJB is created and packaged, you will be ready to deploy and use one.

How Do I Create an EJB?

You will create specific types of EJB as you progress through the book. However, the creation of EJBs follows the same steps and principals for all types of EJB.

The Creation Mechanism

As you may have gathered from the previous discussion on EJB contents, the EJB developer must go through the following cycle:

1. Design and define the business interface. This may involve mapping from a UML model of the solution into Java.
2. Decide on a bean type appropriate to the task in hand. Entity, Session, and Message-driven beans all have their own pros and cons. If you choose to use a Session bean, another question is whether to use a stateful Session bean or a stateless Session bean. Choice of the appropriate type is discussed in more detail on Days 5, 6, 7, and 10.
3. Decide which home interface methods are appropriate for the bean type and define the home interface for the EJB.
4. Create (or generate) a “boilerplate” bean with correct lifecycle methods.
5. Create your business logic by filling out the business methods.
6. Fill out lifecycle methods to control creation, destruction and to manage state (if applicable).

If your EJB classes are written correctly, all that remains is to wrap them up as a deployable unit. However, there are certain caveats you should bear in mind while creating your bean.

Caveats on Code Creation

Due to the managed nature of the bean lifecycle, the EJB container imposes certain restrictions on the bean including:

- EJBs cannot perform file I/O. If you need to log messages or access files, you must find an alternative mechanism.
- EJBs are not allowed to start threads. All threading is controlled by the container.
- EJBs cannot call native methods.
- EJBs cannot use static member variables.
- There is no GUI available to an EJB, so it must not attempt to use AWT or JFC components.
- An EJB cannot act as a network server, listening for inbound connections.
- An EJB should not attempt to create classloaders or change factories for artifacts, such as sockets.
- An EJB should not return `this` from a method. Although not strictly a restriction (the container will not prevent you from doing it), it is identified as being a very bad practice. This relates to the earlier discussion that a bean should not implement its associated remote interface. This would potentially give a client a direct remote reference to the bean rather than the EJBObject. Instead, the bean should query its EJB context for a reference to its associated EJBObject and return that to the caller.

For a full list of restrictions, see section 24.1.2 of the EJB 2.0 specification (available online at `http://java.sun.com/products/ejb/docs.html`).

Create the Deployable Component

One alternative definition of a component is "a unit of deployment." Following this theme, a component should

- Contain all the information required to deploy it, above and beyond the classes. This is the metadata discussed earlier.
- Be bound up in such a way that it can easily be transported and deployed without losing any parts along the way.

Consequently, after the classes and interfaces for an EJB have been created, the following steps must be performed:

1. Capture the EJB's metadata in a universally understood format. This takes the form of an XML-based deployment descriptor (DD).
2. Bundle the classes and deployment descriptor up in a deployable format, namely a JAR file.

The Deployment Descriptor

The EJB specification defines a standard format of an XML deployment descriptor document that can house EJB metadata. The exact format of a deployment descriptor is usually hidden behind tools that manipulate them on your behalf. However, it is worth examining some of the contents of a deployment descriptor to see how the EJB fits together and how extra information and metadata is provided.

Listing 4.5 shows the deployment descriptor for the example Agency EJB.

LISTING 4.5 Agency Bean EJB Deployment Descriptor

```
 1:  <?xml version="1.0" encoding="UTF-8"?>
 2:
 3:  <!DOCTYPE ejb-jar PUBLIC
➥ '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
➥ 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
 4:
 5:  <ejb-jar>
 6:    <display-name>Simple</display-name>
 7:    <enterprise-beans>
 8:      <session>
 9:       <display-name>Agency</display-name>
10:       <ejb-name>Agency</ejb-name>
11:       <home>agency.AgencyHome</home>
12:       <remote>agency.Agency</remote>
13:       <ejb-class>agency.AgencyBean</ejb-class>
14:       <session-type>Stateless</session-type>
15:       <transaction-type>Bean</transaction-type>
16:       <env-entry>
17:         <env-entry-name>AgencyName</env-entry-name>
18:         <env-entry-type>java.lang.String</env-entry-type>
19:         <env-entry-value>J2EE in 21 Days Job Agency</env-entry-value>
20:       </env-entry>
21:       <security-identity>
22:         <description></description>
23:         <use-caller-identity></use-caller-identity>
24:       </security-identity>
25:       <resource-ref>
26:         <res-ref-name>jdbc/Agency</res-ref-name>
27:         <res-type>javax.sql.DataSource</res-type>
28:         <res-auth>Container</res-auth>
29:         <res-sharing-scope>Shareable</res-sharing-scope>
```

LISTING 4.5 Continued

```
30:       </resource-ref>
31:     </session>
32:   </enterprise-beans>
33: </ejb-jar>
```

The essential parts of the deployment descriptor in Listing 4.5 are

- The `<session>` tag delimits the definition of the `Agency` EJB and indicates that it is a Session EJB (lines 8 and 31).
- The `<ejb-name>` tag defines the name of the EJB, in this case `Agency` (line 10).
- The home and remote interface types (as defined by their fully-qualified class file-names) are specified by the `<home>` and `<remote>` tags, respectively (lines 11–12). The type of the bean itself is defined by the `<ejb-class>` tag (line 13).

In addition, two other parts are of particular note at this point in time:

- An environment entry is defined between lines 16 and 20 by using the `<env-entry>` tag. This indicates that a `String` property called `AgencyName` should be made available to the bean. The value of the property is `J2EE in 21 Days Job Agency`. The environment defined in the deployment descriptor is made available through JNDI under the name `java:comp/env`. In this case, the agency name can be retrieved by looking up the name `java:comp/env/AgencyName`. This lookup can be seen in the `ejbCreate()` method of Listing 4.3.
- An external resource is defined in lines 25–30 using the `<resource-ref>` tag. This defines that a `DataSource` should be made available to this EJB under the name `jdbc/Agency`. As with the environment entry for the agency name, this resource is made available through JNDI under `java:comp/env`, so the EJB can retrieve the `DataSource` by looking up the name `java:comp/env/jdbc/Agency`. Again, this lookup can be seen in the `ejbCreate()` method of Listing 4.3.

It is important to realize that the name used for a `<resource-ref>` is only a logical name. In other words, it is just a text string used by a component to reference an external resource. In theory, the resource name used by the EJB to refer to the data source could be anything (`foo`, for example) as long as it ties in with the information in the deployment descriptor. However, by convention, such names are kept in line with the name you would expect to use under JNDI. As a result, in this example, the data source resource is referred to by the bean as `jdbc/Agency` and will be registered under JNDI with the same name.

All of the EJB classes and the deployment descriptor should then be bundled up in a JAR file. The deployment descriptor should be named `ejb-jar.xml`. If there are multiple EJBs packaged in the same JAR file, the deployment descriptor will have multiple EJB definitions in it. This JAR file is then termed an EJB-JAR file to denote its payload. The JAR file itself can be called anything (within reason) and has a `.jar` file extension.

The EJB-JAR file can also contain any extra resources required by the EJB, such as application-specific configuration information that does not fit in a deployment descriptor environment entry.

Enterprise Applications

Although the EJB-JAR file is now complete, it must form part of an application to serve a useful purpose. J2EE defines that enterprise applications can be built from components (Web, EJB, and application components). The key is how to define the relationships between the different parts of the application—there must be some way of plugging things together.

The answer is that there must be a description of the application itself, which components it uses, how those components relate to each other, and which specific resources they use. This is the information provided by the Application Assembler and Deployer roles.

To provide this information to the target J2EE platform, another level of deployment descriptor is used —the J2EE deployment descriptor. The J2EE deployment descriptor provides the following:

- A list of the components in the application
- Security role information
- Web root information for Web components

This information is stored in an XML file called `application.xml`. All of the constituent component JAR files (such as EJB-JARs) and the J2EE deployment descriptor are then bundled up in another JAR file, this time called an Enterprise Archive (EAR) file, which has a `.ear` extension. The contents of the J2EE deployment descriptor will be covered in more detail as you examine the different parts of the example enterprise application.

Is the application now ready to deploy? Unfortunately, the answer is “Not quite yet.” The J2EE deployment descriptor does not cover information about how to map the application onto a specific J2EE application server, specifically

- The JNDI name under which the application server will make the EJB available. In the case of the `Agency` bean, this would mean that an entry was required to map the bean name of `Agency` to the JNDI name under which the EJB is registered, for example, `ejb/Agency`.
- Information about how the security roles defined map to underlying security principals (this is covered on Day 15).

So, yet another XML-based deployment descriptor is required to contain this information, this time an application server-specific one. This file contains extra mapping information, as previously described, and also any other container-specific information required for a smooth deployment in that environment. This extra deployment descriptor is also stored in the EAR file, ready to be accessed when the application is deployed.

The specific deployment descriptor for the J2EE Reference Implementation (RI) server is called `sun-j2ee-ri.xml`.

How Do I Deploy an EJB?

After an EJB is packaged, it can be deployed in an appropriate J2EE server. There is no limit to the number of times an EJB can be deployed as a part of different applications.

Remember that J2EE defines a separate role for the application deployer. It may be that for particular installations, databases, or other resource names need to be changed to match the local environment. When configuring the application, the deployer can alter this EJB or enterprise application metadata.

Plugging into the Container

When an EJB is deployed into a particular EJB container, the EJB must be plugged into that container. To do this, an EJBObject must be generated based on the EJB's remote interface. This EJBObject will be specific to that EJB container and will contain code that allows it to interface with that container to access security and transaction information. The container will examine the metadata supplied with the EJB to determine what type of security and transaction code is required in the EJBObject.

The container will also generate the home interface implementation so that calls to create, find, and destroy EJB instances are delegated to container-defined methods.

The container will examine the EJB and enterprise application metadata and hook up resource references. It will also provide an environment for the application components.

Finally, the container will register the home interface of the EJB with JNDI. This allows other application components to create and find EJBs of this type.

Performing the Deployment

As mentioned previously, when deploying an EJB or enterprise application, the application developer taking on the J2EE role of deployer can choose to alter certain of the metadata relating to the configuration of the application. Although this can be done manually, it is usually done through a GUI tool to make things easier and to keep things consistent.

After the EJB has been deployed, any subsequent changes to its functionality will mean that the EJB must be re-deployed. If the enterprise application or EJB is no longer needed, it should be undeployed from the container.

How Do I Use an EJB?

Given that EJBs are middle-tier business components, they are of little use without a client to drive them. As mentioned earlier, those clients can be Web components, standalone Java clients, or other EJBs.

Regardless of the type of client, using an EJB requires the same set of steps—namely, discovery, retrieval, use, and disposal. These steps are covered in the next three sections.

Discovery

To create or find an EJB, the client must call the appropriate method on the EJB's home interface. Consequently, the first step for the client is to get hold of a remote reference to the home interface. On Day 3, you looked at naming services and how these can be used to register information in a distributed environment. In a J2EE environment, such a naming service is accessible through JNDI and can be used to store references to EJB home interfaces.

The EJB container will have registered the home interface using the JNDI name specified during deployment (as part of the deployment descriptor). This is the name that the client should use to look up the home interface. Recall from the EJB deployment descriptor shown in Listing 4.5 that the EJB name specified was `Agency`. When deploying the EJB, the deployer has a chance to set the JNDI name by which clients will find this EJB. In this case, you would expect the deployer to simply set a JNDI name of `ejb/Agency` so that the client could find the home interface by looking up `java:comp/env/ejb/Agency`. The following code shows the initial lookup required:

```
try
  {
    InitialContext ic = new InitialContext();
    Object lookup = ic.lookup("java:comp/env/ejb/Agency");
AgencyHome home =
          (AgencyHome)PortableRemoteObject.narrow(lookup, AgencyHome.class);
    ...
  }
  catch (NamingException ex) { /* Handle it */ }
  catch (ClassCastException ex) { /* Handle it */ }
```

As you can see, because the reference returned from JNDI is just an object, you must narrow it to the home interface type you expect—in this case, `AgencyHome`. If there are any problems with the JNDI access or if the wrong object type is returned, a `NamingException` or `ClassCastException` will be thrown.

There is no magic here. The object returned by the JNDI is simply an RMI remote object stub. This stub represents the home interface remote object created by the container when the EJB was deployed. This can be seen in Figure 4.2.

Now that you have a reference to the home interface, you can create the EJB you want to use.

Retrieval and Use

You can now call the `create()` method you saw defined on the `AgencyHome` interface in Listing 4.4 as follows:

```
  try
  {
    ...
    Agency agency = home.create();
    System.out.println("Welcome to: " + agency.getAgencyName());
    ...
  }
  catch (RemoteException ex) { /* Handle it */ }
  catch (CreateException ex) { /* Handle it */ }
```

The `create()` method returns a remote reference to the newly-created EJB. If there are any problems with the EJB creation or the remote connection, a `CreateException` or `RemoteException` will be thrown. `CreateException` is defined in the `javax.ejb` package, and `RemoteException` is defined in the `java.rmi` package, so remember to import these packages at the top of your client class.

Now that you have a reference to an EJB, you can call its methods. The previous code sample shows the `getAgencyName()` method being called on the returned `Agency` reference. Again, whenever you call a remote method that is defined in an EJB remote interface, you must be prepared to handle `RemoteException`s.

You will see later that some types of EJB are found rather than created. In this case, all steps are the same except that the `create()` method is replaced by the appropriate finder method and find-related exceptions must be handled. You still end up with a remote reference to an EJB. All of this is covered later when Entity EJBs are discussed on Day 6.

Disposing of the EJB

You have now created and used an EJB. What happens now? Well, if you no longer need the EJB, you can get rid of it in exactly the same way that you would get rid of a local Java object or a remote Java object defined using RMI—by setting its reference to `null` as follows:

```
// No longer need the agency EJB instance
agency = null;
```

When the local RMI runtime detects that the remote object no longer has any local references, it will trigger remote garbage collection for that object, which means that its remote reference will time out. This will result in the object being de-referenced at the server-side. In the case of the simple `Agency` bean (a stateless Session bean), this will cause the bean to be destroyed.

Although it is possible to use the `remove()` method to get rid of the EJB, you would not normally use this for such a simple bean. Use of this method is discussed in more detail on Days 5 and 6.

Running the Client

You are now in a position to write a simple application client for the `Agency` EJB. After you have written it, you will want to compile and run it.

Before compiling your client, you should ensure that you have `j2ee.jar` on your classpath. This JAR file lives in the `lib` directory under `J2EE_HOME`. If you are using an enterprise IDE, you may find that all the relevant classes are already in your classpath.

To compile and run the client, you will need the following:

- The J2EE classes. These must be accessible through the classpath.
- Access to the EJB's home and remote interface class files via the classpath.
- RMI stubs for the home and remote interfaces. These can either be installed on the local classpath or downloaded dynamically from the EJB server.

- If the client does not have the JNDI name of the EJB compiled in, you may want to provide this on the command line or through a system property.

When you deploy the EJB, you should be able to ask the container for a client JAR file. This client JAR file will contain all of the classes and interfaces needed to compile the client (as defined in the previous bulleted list). You should add this client JAR file to your classpath when compiling your client.

In theory, this should be it. However, you will find that any form of security definition on the server will require you to authenticate yourself before you can run the application. In this case, you must explicitly use the client container to provide the required security mechanism.

The client container is called `runclient` under the J2EE RI.

Deploying and Using an EJB in the J2EE Reference Implementation

You should now be in a position to write and test an EJB client. However, before you can do that, you must deploy an EJB that it can use. In this section, you will look at how to deploy an EJB in the J2EE Reference Implementation (RI) and how to then use it from a simple client.

The J2EE on which your EJB is deployed will provide a complete server-side environment. It houses any EJBs, runs a Web Server for JSP/servlets, runs a naming server for storing component location information, and provides database access. All J2EE-compliant application servers will do this—even a non-commercial version, such as the J2EE RI. The RI also provides you with a ready-to-use database so you do not have to concern yourself with hooking up to an existing database or installing a separate one.

To deploy and test your EJBs (and servlets/JSPs later), you only need a single machine. Both the J2EE client and the J2EE server (and its EJBs, servlets and JSPs) can run on the same machine. No connection to the Internet is required. The J2EE RI is available on multiple platforms (Win32, Solaris, and Linux) and should be consistent across these platforms, so that J2EE applications created on one platform can be deployed on another.

If you encounter problems at any stage, try referring to the troubleshooting section just before today's Summary.

Before running any of the tools described in this section, you will need to set the `J2EE_HOME` environment variable to the location on your hard drive where you deployed the J2EE reference implementation. You should also add the `bin` directory below `J2EE_HOME` to your executable search path (`%PATH%` under Windows or `$path` under Unix/Linux) so that you can run J2EE tools and batch files from the command line.

Opening the Case Study EAR File

To deploy and manipulate EJBs under the RI, you will use a graphic tool called `deploytool`. Before you start using this, you will need to do two things:

1. Ensure that you have created and configured your database environment as described on Day 2.
2. Start the J2EE RI runtime environment and the associated `Cloudscape` database. To do this, run the `cloudscape` and `j2ee` scripts/batch files found in the `bin` directory under `J2EE_HOME` as follows:

   ```
   cloudscape -start
   j2ee -verbose
   ```

   The use of the `-verbose` flag for J2EE is not strictly necessary, but you may find it useful to help you understand what the J2EE server does when it starts up.

Now you are ready to run the `deploytool`. Again, this is a script/batch file found in the `bin` directory under `J2EE_HOME`. When you run it, the GUI screen will appear as shown in Figure 4.3.

You should now be able to open the initial agency enterprise archive provided in the JAR subdirectory of the Day 4 exercise code on the CD-ROM (`agency.ear`). Do this through the menus by selecting File, Open and then browsing for the file in the subsequent Open Object dialog box. Select the EAR file and click the Open Object button. The agency application will now be displayed in the list of applications, as shown in Figure 4.4.

All of the code for the Agency EJB that is contained in the `Agency` application can be found in the `agency` subdirectory of the `src` directory under the Day 4 Exercise part of the CD-ROM.

Now that the enterprise application is loaded in `deploytool`, you can examine its settings.

FIGURE 4.3

The initial screen shown by `deploytool`.

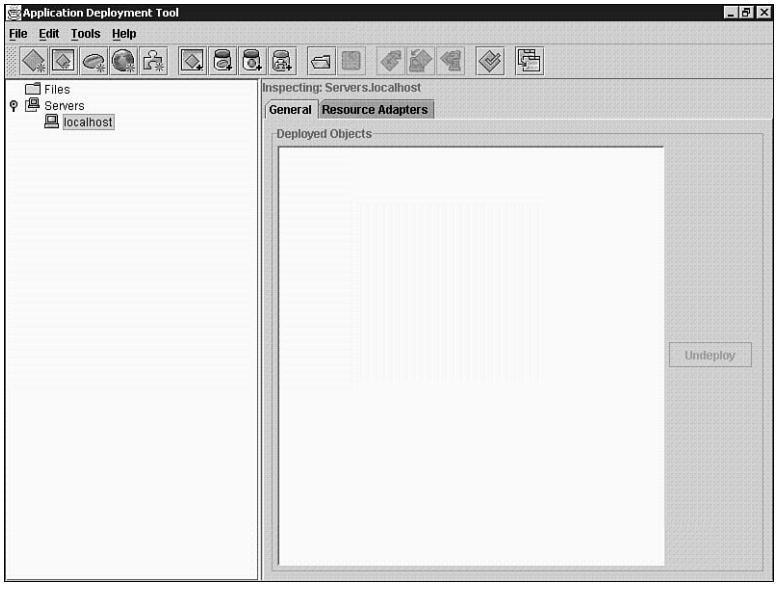
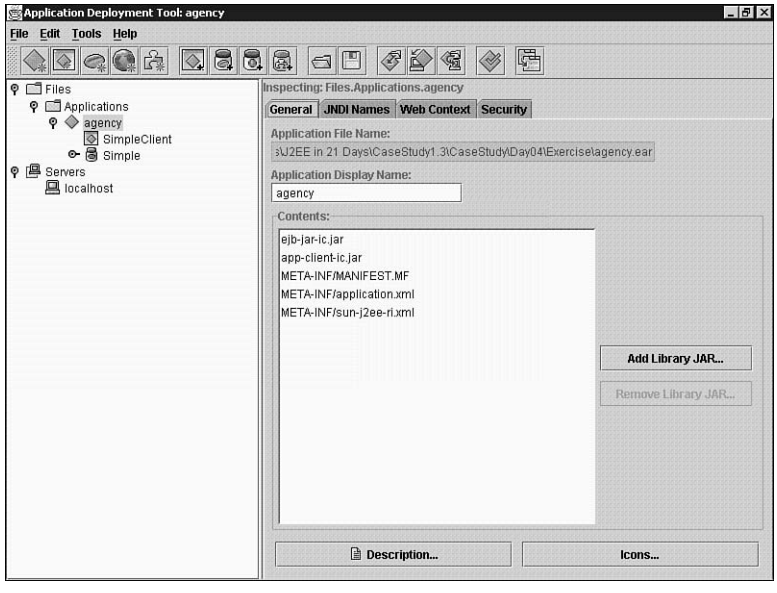


FIGURE 4.4

The `Agency` application has now been loaded by `deploytool`.



4

Examining the Case Study Application

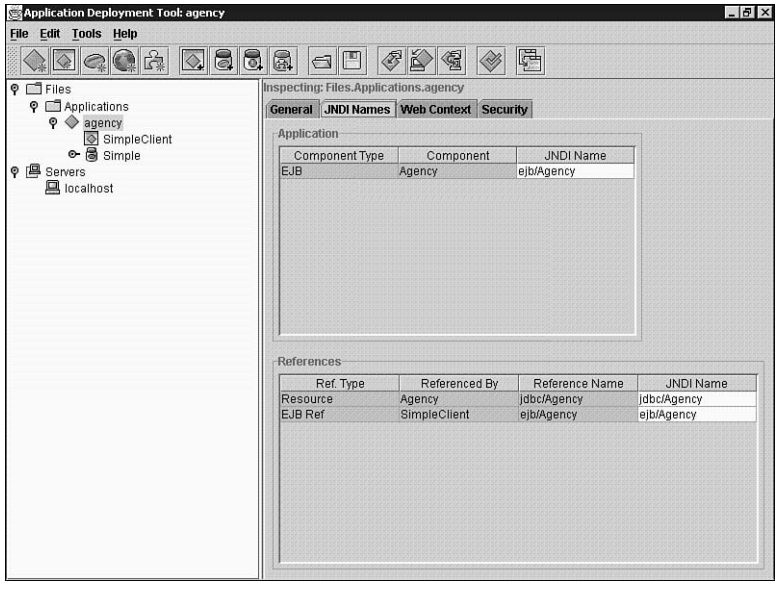
You can use `deploytool` to examine and alter deployment descriptor information for the application and, if necessary, for individual components, such as EJBs.

If you select the JNDI Names tab for the `agency` application, you will see information about the resources that the application exports and consumes. This is largely based on information defined in the application deployment descriptor and the container-specific deployment descriptor described earlier in the "Enterprise Applications" section.

In Figure 4.5, you can see in the Application box that there is a single EJB in this initial form of the application. That EJB can be referenced through JNDI using the name `ejb/Agency`.

FIGURE 4.5

`deploytool` displays the JNDI information from the `Agency` application deployment descriptor.



In the References box, you can see that two of the components in the application use external resources. First, you can see that the component named `Agency` (the EJB) uses a resource called `jdbc/Agency` that is registered under JNDI as `jdbc/Agency`.

The References box also indicates that the application client, `SimpleClient`, references the `Agency` EJB by using the name `ejb/Agency` that appears under JNDI as `ejb/Agency`.

You can also examine the settings of the EJB through `deploytool`. Click the icon next to the `Simple` JAR file symbol to show the EJBs contained in the `Simple` EJB-JAR file. There is a single EJB in the JAR file called `Agency`. If you select the `Agency` EJB, you will see the properties defined in the deployment descriptor for that EJB. Select the Resource Refs tab to see what external resources this EJB uses, as shown in Figure 4.6.

FIGURE 4.6

You can examine the deployment descriptor information for a single EJB, such as the external resources it expects.

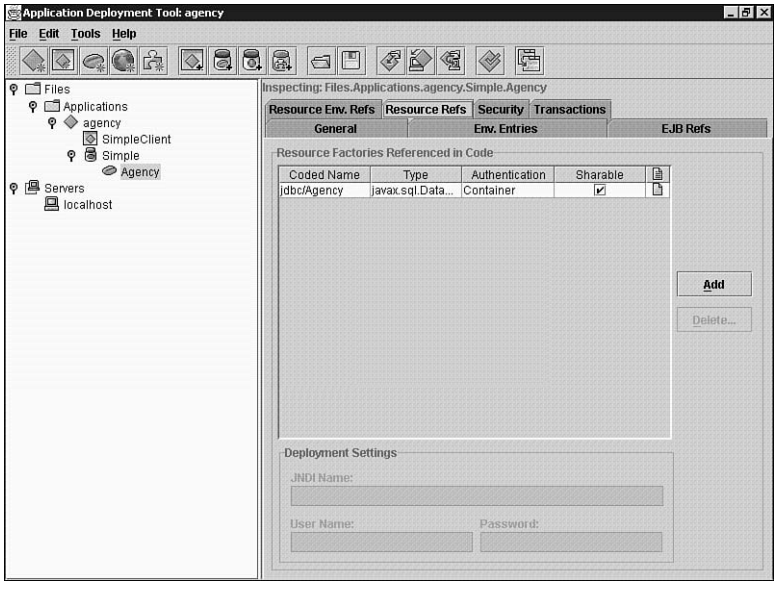
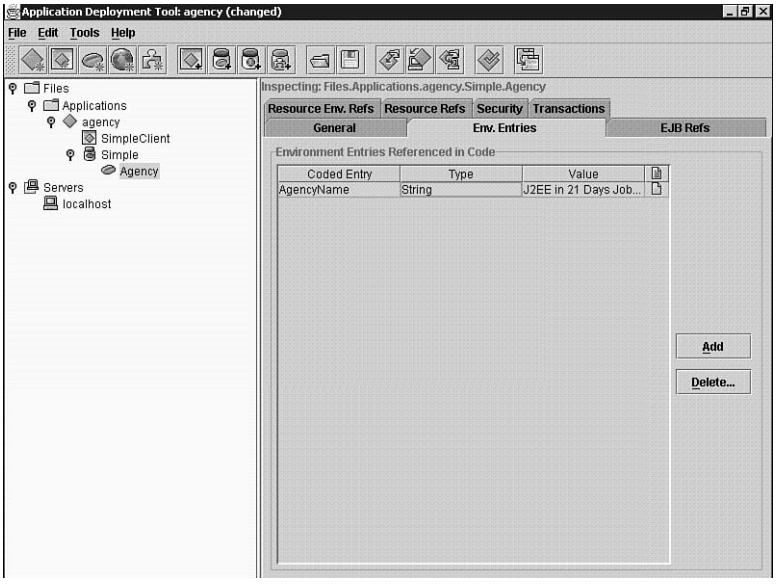


Figure 4.6 shows that the `Agency` EJB expects one resource called `jdbc/Agency` that is of type `javax.sql.Datasource`. This is the EJB deployment descriptor information you saw in Listing 4.5.

Figure 4.7 shows the environment entries for the `Agency` EJB. If you want to alter the `AgencyName` defined there, you can just double-click the Value field and type in an alternative name. If you make any changes to the configuration of the application or any of its components, the suffix `(changed)` will be added to the application name in the title bar.

FIGURE 4.7

Environment entries can be viewed or edited through `deploytool`.



Deploying the Case Study Application

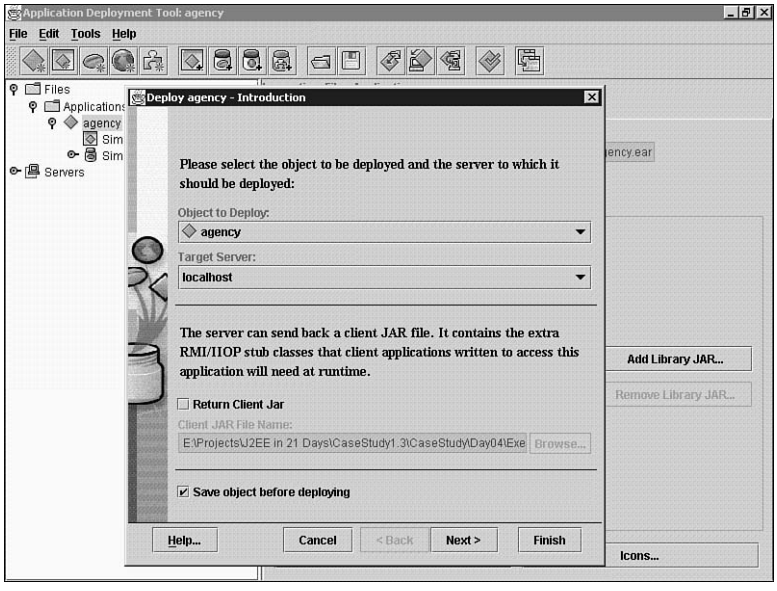
You can deploy the server-side components of the `agency` application (in this case, a single EJB) using `deploytool`. To deploy them, select the `agency` application item (indicated by a blue diamond under the `Applications` folder in the explorer area on the left side), and select Tools, Deploy from the `deploytool` menu. This will display the initial deployment screen shown in Figure 4.8.

As you can see from Figure 4.8, the default target host is `localhost`. This is fine, presuming that your copy of the J2EE RI is running on the local machine. If not, you should Cancel and add the appropriate server name through the File, Add Server menu item before proceeding.

The other point to note for this screen is that it allows you to obtain a client JAR file. Recall that this client JAR file will contain all of the classes required by a client of the application being deployed. A pre-prepared client JAR file is provided in the `jar` subdirectory of the Day 4 exercise code on the CD-ROM (`agencyClient.jar`), but you will need to obtain a client JAR file for any new applications or components you deploy. If you check the Return Client Jar box, you can browse and select an appropriate location to store the returned JAR file.

FIGURE 4.8

You can select a server on which to deploy an enterprise application.



Caution

If you are deploying the agency application on an other machine (not `localhost`), you should not use the pre-provided `agencyClient.jar` file. Instead, select the option to return the client JAR file and add this JAR to the classpath when you run the client later.

Click Next to move on to the `JNDI Names` screen. This gives the deployer another chance to provide server-specific deployment information. Note that this is the same information you saw earlier when examining the application's JNDI information. For now, just click the Next button. At the last screen, click Finish to deploy the application to the selected server.

The progress of the deployment is shown in a separate window, as seen in Figure 4.9. The blue and green bars are progress bars that will increase as the deployment proceeds. Click OK when the deployment is complete.

When you are done, the `agency` enterprise application should have been deployed, as shown in Figure 4.10. To view the applications deployed on a server, expand the Servers folder in the explorer area on the left side and then expand the particular server, such as `localhost`.

FIGURE 4.9

`deploytool` will show you the progress of the deployment.

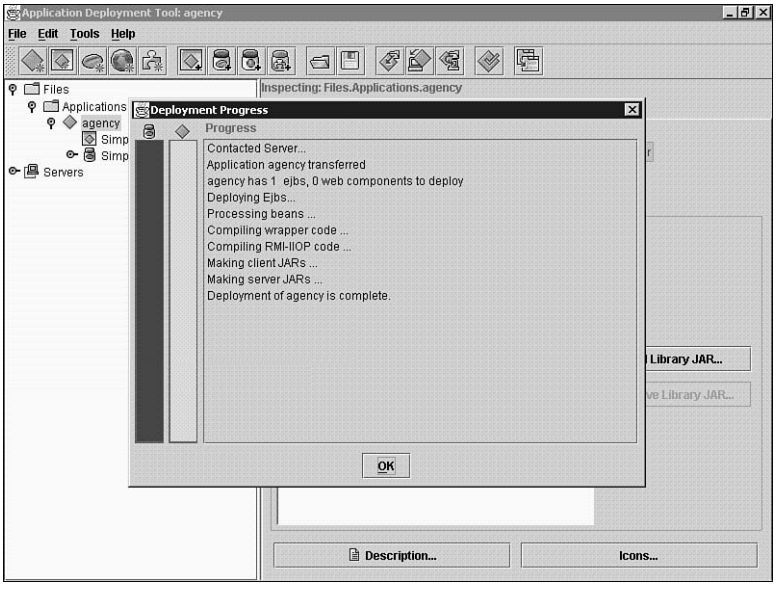
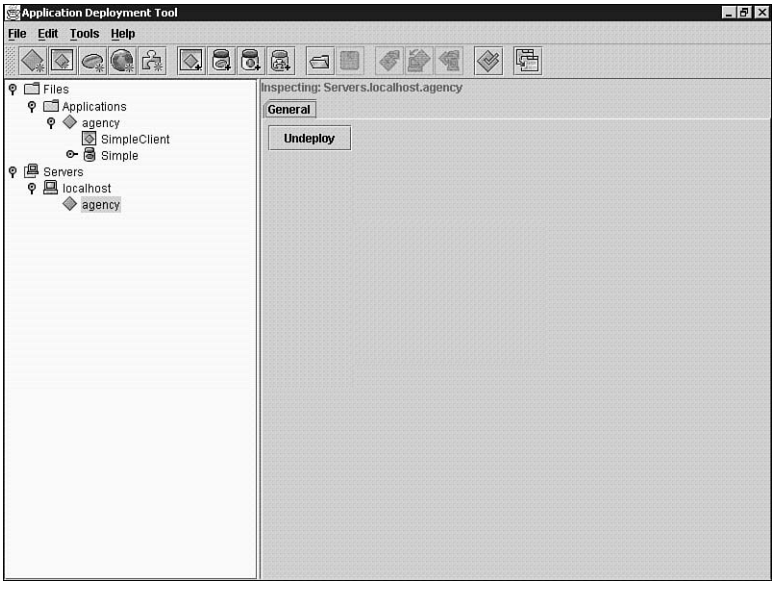


FIGURE 4.10

You can list the applications deployed on a server.



Testing the Case Study Application

After you have successfully deployed the server-side components of the application under the J2EE RI, you can run the test client to check that everything is okay.

The test client, `client.SimpleClient`, is provided pre-compiled in the `classes` subdirectory. To run this, use the script/batch file `runSimple` that can be found in the `run` subdirectory. When you run the client, you will need to provide a username and password. Use the username `guest` with a password of `guest123`. Your interaction with the client should look something like the following:

```
Initiating login ...
Username = null
Enter Username:guest
Enter Password:guest123
Binding name:`java:comp/env/ejb/Agency`
Welcome to: J2EE in 21 Days Job Agency
Customer list: J2EE in 21 Days Job Agency
abraham
alfred
george
winston
Unbinding name:`java:comp/env/ejb/Agency`
```

If you are not able to run this test, refer to the "Troubleshooting the Case Study Application," section next.

As you can see, this has used the `Agency` EJB to list all of the customers in the job agency database. The code for the client is provided in the `client` sub-directory of the `src` directory. If you examine it, you will see that it is identical to the code shown earlier, but with some extra code to invoke the customer listing methods on the EJB and display the results.

If you examine the `runSimple` script/batch file, you will see that it uses the `runclient` utility provided by the J2EE RI as follows:

```
runclient -client agency.ear -name SimpleClient -textauth
```

The precise command line used will differ between platforms to define the location of the agency EAR file using the correct direction of slashes (backslash or forwardslash). However, this makes no difference in how the command works.

This runs the client application directly from the EAR file without having to unpack it. The command line specifies that the client is called `SimpleClient` and that it lives in the `agency.ear` archive. It also specifies that simple, text-based authentication should be used between the client and the server.

The other thing to note in the script/batch file is that the environment variable `APPCPATH` is set before `runclient` is called. The `runclient` utility uses this environment variable to find the client JAR file for the application (in this case, it points to `agencyClient.jar`). This information will be needed at runtime, not only because the EJB's interface class files are needed, but also because it also contains RMI stubs that are targeted at the correct server and deployment information that is used to map resource names to the target server's JNDI names.

Should you need to re-compile the client, or to compile your own client, you will need to add this client JAR file to your classpath.

Initial Naming Context for the Client

When creating a JNDI `InitialContext`, the client runtime must get hold of information regarding the content of this initial context. In the case of `runclient`, it will find mappings in the deployment descriptors contained in the client JAR file. These mappings will be used to set up the initial context for the client. Consequently, when the client gets a new initial context and looks up `java:comp/env/ejb/Agency`, for example, the local naming service runtime will be able to use the pre-provided mapping information to match this text string to the appropriate JNDI name.

Other application servers may use different mechanisms to set up this initial context, such as passing a `Properties` object to the naming runtime containing the appropriate information.

Troubleshooting the Case Study Application

If you have problems running the case study application, check out the following possible issues:

- Have you started the J2EE RI (`j2ee -verbose`)? Make sure by locating its console window or looking for it in the list of processes or applications on your machine.
- Have you started the cloudscape database (`cloudscape -start`)? Try running the initial database test at the end of Day 2 to ensure that the data is present and that the database server is running.
- Have you deployed the EJBs? By opening the EAR file, you are simply loading the enterprise application into the `deploytool` environment. You must explicitly deploy the application to the server you are using through the Tools, Deploy menu.
- Have you set the `classpath` correctly for your client? The client will need access to the J2EE libraries in order to run.

- Try re-creating the client JAR file when you deploy the J2EE application to your server. Make sure that this client JAR file is on the classpath when you compile and run the client application.
- Check the J2EE RI console window to see if exceptions or errors are shown there.
- Check the J2EE log files under the `logs` directory in `J2EE_HOME`. There is a directory below `logs` that is named after the machine on which the server is running. Below this, there are two nested `j2ee` directories. In the lower of these, you will find various log files that you can examine for errors.

If you still have problems and you suspect that there is a problem with the configuration of your J2EE RI, you can either re-install the RI or you could try deleting the server-specific repository directory and then re-starting your server. You will lose all of your deployed J2EE applications, but you may find this easier than re-installing. Under the `J2EE_HOME` directory, you will find a directory called `repository`, and below this there will be a directory named after the server on which you are running this instance of the RI (for example, if your hostname is "fred", there will be a `fred` directory below `repository`). Stop the J2EE RI, remove the directory that is named after your server, and then start the J2EE RI again.

Summary

Today, you have seen common ways that EJBs are used in applications and why you would want to use them. You have seen that an EJB will have a home interface, a business or remote interface, and an implementation. You have seen how the EJB container will provide much of the underlying code to support the EJB, and that it relies on detailed deployment information that defines the EJB's requirements.

You have also seen that a J2EE application consists of components and deployment information and how the server-side part of such an application can be deployed. You have seen a client that is able to use such server-side components and the code required to write such a client.

Q&A

Q How many Java classes and interfaces must I write to create an EJB?

A The EJB writer must define a remote (or business) interface, a home interface, and the bean implementation itself.

Q Why does an EJB run inside a container?

A The container provides many services to the EJB, including distribution, lifecycle, naming/registration, transaction management, security/authentication, and persistence. If the container did not exist, you would have to write all the code to interact with these services yourself.

Q What issues surround passing an object as part of a remote method call?

A To be passed as an argument or return type, an object must be either serializable or remote. If it is neither of these, an error will occur at runtime. If an object is defined as serializable, a new copy will be created and passed/returned. This can add to the overhead of making the method call, but it is a very useful tool when trying to cut down the amount of network traffic between clients and EJBs (as you will see later on Day 18).

Q Most of the deployment descriptor information is straightforward, but what is the difference between a `<resource-ref>` and an `<env-entry>`, and what sort of information is contained in each type of entry?

A A `<resource-ref>` is part of a deployment descriptor that defines an external resource used by a J2EE component. The `<resource-ref>` will define a name and type for a resource together with other information for the container. The information in a `<resource-ref>` is really for the container rather than for the EJB itself. To access a resource defined in a `<resource-ref>`, you would use JNDI to look up its name `java:comp/env/jdbc/Agency`.

On the other hand, an `<env-entry>` contains information that is intended for the EJB itself rather than the container. It will define a name, a class type and a value. The contents of `<env-entry>` elements are usually strings. Again, you would use JNDI to look up its name, `java:comp/env/AgencyName`.

Exercises

The intention of this day is for you to familiarise yourself with the EJB environment and the use of EJBs. To ensure that you are comfortable with these areas, you should attempt the following tasks.

1. If you have not already done so, follow the steps to deploy the example `Agency` EJB from the Day 4 `Exercise` directory on the CD-ROM.
2. Examine the information displayed by `deploytool` and make sure that you can identify where the resource reference for the `Agency` JDBC connection is set, where the environment reference for the agency name is set, and where the JNDI name of the `Agency` EJB itself is set.

3. Use the `runSimple` script/batch file provided under the Day 4 `Exercise` directory on the CD-ROM to run the test client. Make sure that this client runs without errors and successfully lists all the customers in the agency database.
4. Without referring to the example client (but referring to the material you have covered today), create your own simple test client for the `Agency` EJB from scratch. This should just consist of a command-line client that creates an instance of an `Agency` EJB and asks it for its name.
5. Try changing the name under which the EJB is registered in JNDI using `deploytool`. Change the JNDI name used by your client to find the `Agency` EJB and make sure that it still works.

WEEK 1

DAY 5

Session EJBs

On Day 4, "Introduction to EJBs," you learned that business functionality can be implemented using Session beans, and you deployed a simple Session bean into the EJB container. Today, you will learn

- The uses of Session beans in more detail
- The different Session bean types and how to specify, implement, and deploy both stateless and stateful Session beans
- About common practices and idioms when using Session beans

Overview

Session beans are a key technology within the J2EE platform because they allow business functionality to be developed and then deployed independently of the presentational layer.

For example, you might create an application with a user interface built using Java's Swing API. This application might then provide access to some business functionality for the employees working on the company's internal network.

If the underlying business functionality is implemented as Session beans, a different user interface could take its place without having to redevelop the entire application. A Web-based interface would make the application available from the Internet at a single stroke.

There are two types of Session beans, and a couple of analogies help explain the differences between them. You almost certainly will have used the so-called wizards—helpers to guide you through some task—in any modern word-processing program or IDE. A wizard encapsulates a conversation between you the user and the application running on the computer. The steps in that conversation are dictated by the Next and the Back buttons. The wizard remembers the answers from one page, and these sometimes dictate the choices for the next. When you are done, you select the Finish button and the wizard goes away and does its stuff.

The wizard is analogous to a *stateful* Session bean. The wizard remembers the answers from each page, or put another way, it remembers the state of the conversation. It also provides some service, as characterized by the Finish button. This is precisely what a stateful session bean does.

Here is another analogy, this time with databases. You may well have had cause to write stored procedures. These are named routines (methods and functions) that are written in a database vendor's version of the SQL language (for example, PL/SQL for Oracle and Transact-SQL for Microsoft SQL Server) and stored in the database. They provide a way to implement business rules on the database.

To invoke a stored procedure, a client-side application needs to know just the name of the stored procedure and the parameters it requires. No knowledge of the underlying database schema is needed, so to call a stored procedure called `find_jobs_by_advertiser` written in Transact-SQL, the client would use the following:

```
exec find_jobs_by_advertiser "winston"
```

Behind the scenes, this would probably run a query against the `Job` table, but the important thing is that the client does not need to know this detail.

A stored procedure is analogous to a stateless Session bean. The stored procedure just provides a service and can be invoked by any client. You may be wondering why have Session beans at all if stored procedures—which are a tried-and-trusted technology—already solve the problem. But Session beans do have a number of advantages. Implementing stateful conversations is cumbersome using stored procedures, but trivial with Session beans. Also, stored procedures are written in some database vendor's proprietary dialect of SQL, so they are not portable across RDBMS. Session beans are, of course, written in Java, so they will be portable across any compliant EJB container.

Session beans provide a service to a client application. In other words, Session beans are an extension of a client's business functionality into the middle tier.

The Unified Modeling Language offers a number of useful diagrams to help specify an application, one of which is the use case diagram (see Appendix A, "An Introduction to UML," found on the CD-ROM accompanying this book, for further details).

Each use case represents an item of business functionality that is required to fulfill an end-user's goal. However, while use case diagrams indicate the system boundary of the application being developed, the use cases themselves typically say nothing about the actual look-and-feel of the system. In other words, the presentational layer or user interface is not directly specified.

It is quite possible to directly relate UML use cases to Session beans. A given use case specifies an item of business functionality, while the Session bean implements that functionality. Neither are concerned with the detail of how that functionality is presented to the user.

This direct correspondence of the logical design (as characterized here in UML use cases) to the physical implementation (in this case, Session beans) is one of the reasons that the J2EE platform is so appealing, quickly allowing designs to be realized into working code.

The `javax.ejb` Package for Session Beans

Now it is time to add a little more detail. EJBs are written by implementing various interfaces of the `javax.ejb` package. Some of these are implemented by the bean itself. In other words, this is the code that you, the developer must write. Others are implemented either directly by the EJB container or are implemented by classes generated by the tools provided by your EJB container vendor, such as the J2EE RI.

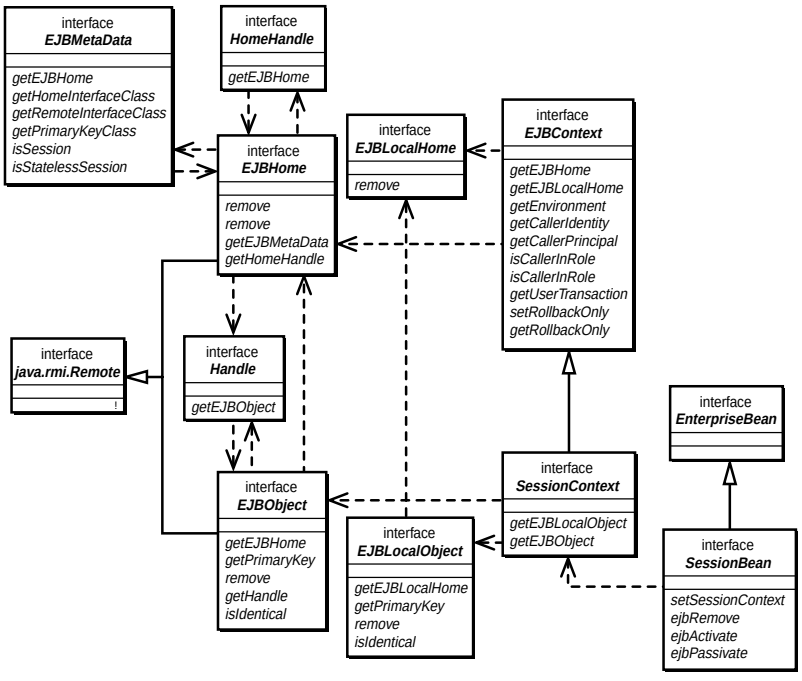
Figure 5.1 shows a UML class diagram of the interfaces in `javax.ejb` that support Session beans.

Central to the EJB architecture are the `javax.ejb.EJBHome` and `javax.ejb.EJBObject` interfaces, common to both Session beans and Entity beans. These both extend the `java.rmi.Remote` interface, meaning that the classes that implement them (not shown) are available through RMI stubs across the network.

The `javax.ejb.EJBLocalHome` and `javax.ejb.EJBLocalObject` interfaces are local equivalents, and the classes that implement these are accessible only locally (that is, by clients that reside within the same EJB container itself). Because local interfaces are most often used with Entity beans, and also because there's plenty for you to learn about today already, there's no major discussion of them until tomorrow.

FIGURE 5.1

The `javax.ejb` *package defines remote and local interfaces, as well as an interface for the Session bean itself to implement.*



The `javax.ejb.EJBContext` interface provides access to the home interfaces and, as you can see from its method list, also provides security and transaction control. The `javax.ejb.SessionContext` subclass is used only by Session beans and provides a reference to the bean's `EJBObject`, that is, its interface for remote clients. Every EJB must have a remote interface (or a local interface, discussed on Day 6, "Entity EJBs").

The `javax.ejb.HomeHandle` and `javax.ejb.Handle` interfaces provide a mechanism to serialize a reference to either a home or a remote interface for use later. This capability is not often used, so isn't discussed further.

The Session bean itself implements the `javax.ejb.SessionBean` interface that defines the bean's lifecycle methods and has an implementation for all of the methods defined in the remote or the home interface.

Stateless Session Bean Lifecycle

You already know that there are two different types of bean—stateful and stateless. You'll be learning about both types today, first, the simpler stateless bean. The `Agency` bean from the case study will be used for the example code.

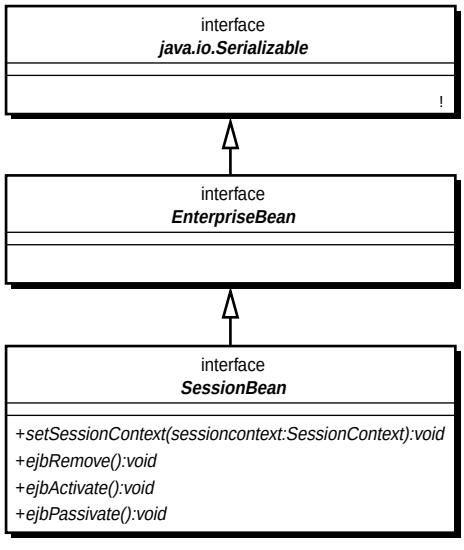
Stateless beans hold no state for any particular client, but they do have a lifecycle—and thus different states—imposed on them by the EJB architecture. Specifically, these are the interactions between the bean and the container in which it has been deployed.

This is a recurrent theme throughout the EJB architecture, so it is important to fully understand it. The methods you define in your bean will be invoked either by its client or by the EJB container itself. Specifically, the methods invoked by the client will be those defined in the remote interface, whereas the methods invoked by the container are those defined by the `javax.ejb.SessionBean` interface. The bean must also provide methods that correspond to the `create` method of the bean's home interface.

Figure 5.2 shows the `SessionBean` interface and its super-interfaces.

FIGURE 5.2

The `javax.ejb.SessionBean` interface defines certain lifecycle methods that must be implemented by Session beans.



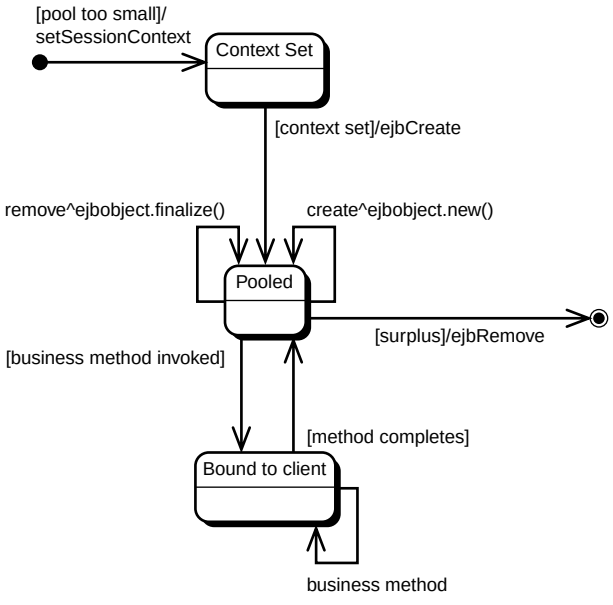
In the case study, the `AgencyBean` class indicates that it is a Session bean implementation by implementing this interface:

```
package agency;

import javax.ejb.*;
// some import statements omitted

public class AgencyBean implements SessionBean
{
    // code omitted
}
```

The lifecycle for Session beans, as perceived by the Session bean and as likely to be enacted by the EJB container, is as shown in the UML state chart diagram in Figure 5.3.

FIGURE 5.3

Stateless Session beans have a lifecycle managed by the EJB container.



The lifecycle is as follows:

- If the EJB container requires an instance of the stateless Session bean (for example, because the pool of instances is too small), it instantiates the bean and then calls the lifecycle method `setSessionContext()`. This provides the bean with a reference to a `SessionContext` object, providing access to its security and transaction context.
- Immediately after the context has been set, the container will call `ejbCreate()`. This means that the bean is now ready to have methods invoked. Note that the `ejbCreate()` method is not part of the `SessionBean` interface, but nevertheless must be declared in the bean.
- When a client invokes a business method, it is delegated by the bean's `EJBObject` proxy to the bean itself. During this time, the bean is temporarily bound to the client. When the method completes, the bean is available to be called again.

The binding of the bean to the client lasts only as long as the method takes to execute, so it will typically be just a few milliseconds. The EJB specification specifies this approach so that the bean developer does not need to worry about making the bean thread-safe.

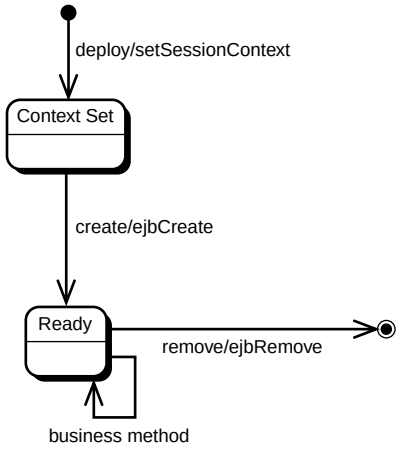
To support the case where two (or more) clients need to invoke the service of some stateless Session bean at the same time, most EJB containers hold a pool of Session beans. In general, the pool will never be larger than the maximum number of concurrent clients. If a container decides that the pool is too large or that some Session bean instances are surplus, it will call the bean's `ejbRemove()` method.

> As you will see on Day 12, "Servlets," servlets have similarities with stateless Session beans. However, in the servlet specification, they are defined to work in precisely the opposite way; by default, a servlet must be thread-safe, and there is only one instance of it.

If the client calls `create()` or `remove()`, the bean itself is not necessarily affected. However, the client's reference to the bean will be initialized (or destroyed). The client is not aware of the complexities of this lifecycle, so the client's perception of a stateless bean is somewhat simpler, as shown in Figure 5.4.

FIGURE 5.4

The client's perception of the bean's lifecycle is simple.



From the client's perspective, the bean is simply instantiated when the client calls `create()` on the bean's home interface, and is removed when the bean calls `remove()` on the bean itself.

The EJB specification does not attempt to prescribe too closely the implementation of EJB containers, and correctly focuses instead on their specification. Unfortunately, it does not always identify the only realistic implementation.

For example, the EJB specification suggests that the EJB container is at liberty to adopt any appropriate pooling policy for Session beans. In Figure 5.3, you saw the state chart for a container using an eager instantiation policy, pre-instantiating beans before they are necessarily used. However, the fact that beans can throw `CreateException` exceptions from their `ejbCreate()` method seems to imply that only a lazy instantiation policy—instantiating beans only as they are required—could be used.

In fact, it *is* the case that some EJB containers do not maintain a pool of Session bean references and, instead, simply instantiate beans as required. In other words, the actual lifecycle for the bean matches that perceived by the client. While this might seem a wasteful approach, in fact it is not; modern JVMs are becoming so efficient that maintaining a pool of beans is more expensive than simply instantiating beans as needed.

Figures 5.3 and 5.4 show how the methods of the `SessionBean` interface are invoked through the bean's lifecycle. You will have noticed that the `ejbActivate()` and `ejbPassivate()` methods are not mentioned; this is because these methods are only called for stateful Session beans, a topic covered later today. However, given that these methods are in the `SessionBean` interface, they do require an implementation. For stateless Session beans, this implementation will be empty.

The implementation of the lifecycle methods is covered later today in the "Implementing a Stateless Session Bean" section.

Specifying a Stateless Session Bean

As you will by now have gathered, the responsibilities of Session beans (and indeed, Entity beans) are specified through its remote and home interfaces. These are what the EJB container makes available to the remote clients.

To define a home interface for a stateless Session bean, extend `javax.ejb.EJBHome`. To define a remote interface, extend `javax.ejb.EJBObject`. Because both `EJBHome` and `EJBObject` extend the `java.rmi.Remote` interface, the rules for remote objects (in the Java sense of the word) must be followed.

The following is the home interface for the `Agency` session bean. If it looks familiar, it should be—you saw this for the first time just yesterday.

```
package agency;

import java.rmi.*;
import javax.ejb.*;
public interface AgencyHome extends EJBHome
{
    Agency create() throws RemoteException, CreateException;
}
```

The `AgencyHome` interface defines a single no-arg method called `create()`. This method returns an `Agency`, which is the remote interface for the `Agency` bean. Because this remote interface is remote (that is, extends `java.rmi.Remote`), what the client that calls this interface will receive is a reference to the remote `Agency` object. In other words, the client will obtain an RMI stub to the `Agency`.

The EJB specification requires that stateless Session beans must define this single no-arg version of the `create()` method. The bean can perform any initialization it requires there. The `create()` method throws `java.rmi.RemoteException`, as required for remote objects, and also throws `javax.ejb.CreateException`. This is an exception that the bean can throw to indicate that it was unable to initialize itself correctly.

The `create()` method in the home interface implies a corresponding `ejbCreate()` method in the bean class itself. This delegation to a method with an `ejb` prefix is prevalent throughout the EJB specification, so you will become quite familiar with it over the next few days. The corresponding code in the `AgencyBean` class is as follows:

```
package agency;

// some import statements omitted
import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public class AgencyBean implements SessionBean
{
    public void ejbCreate() throws CreateException {
        // implementation omitted
    }

    // code omitted
}
```

Note that the `ejbCreate()` method also takes no arguments because the argument list must match. The `throws` clause includes `javax.ejb.CreateException`, because that was defined in the home interface, but does not include `java.rmi.RemoteException`. This is because the bean itself is not remote; it is the code generated by the vendor's deployments tools that is remote. The EJB specification requires also that `ejbCreate()` method returns `void`.

Listing 5.1 shows the remote interface for the Agency session bean. Again, you saw this yesterday:

LISTING 5.1 Remote Interface for the Stateless Agency Bean

```
 1: package agency;
 2:
 3: import java.rmi.*;
 4: import java.util.*;
 5: import javax.ejb.*;
 6:
 7: public interface Agency extends EJBObject
 8: {
 9:     String getAgencyName() throws RemoteException;
10:
11:     Collection findAllApplicants() throws RemoteException;
12:     void createApplicant(String login, String name, String email)
13:         throws RemoteException, DuplicateException, CreateException;
14:     void deleteApplicant (String login)
            ➥throws RemoteException, NotFoundException;
15:
16:     Collection findAllCustomers() throws RemoteException;
17:     void createCustomer(String login, String name, String email)
18:         throws RemoteException, DuplicateException, CreateException;
19:     void deleteCustomer(String login)
            ➥throws RemoteException, NotFoundException;
20:
21:     Collection getLocations() throws RemoteException;
22:     void addLocation(String name)
            ➥throws RemoteException, DuplicateException;
23:     void removeLocation(String code)
            ➥throws RemoteException, NotFoundException;
24:
25:     Collection getSkills() throws RemoteException;
26:     void addSkill(String name)
            ➥throws RemoteException, DuplicateException;
27:     void removeSkill(String name)
            ➥throws RemoteException, NotFoundException;
28:
29:     List select(String table) throws RemoteException;
30: }
```

You can see that the Agency Session bean provides a number of sets of functionality, managing applicants, customers, locations, and skills. These services manipulate data within the database, but there is no underlying state for the bean itself. In each case, the methods throw java.rmi.RemoteException as required and also throw various other exceptions. The DuplicateException and NotFoundException are user-defined exception classes that simply extend java.lang.Exception. You encountered these classes yesterday.

For each of these methods in the remote interface, there is a corresponding method in the Session bean. As was noted before, this is not because the bean has implemented the remote interface (it hasn't) but because the EJB specification requires it so that the `EJBObject` proxy (the vendor-generated implementation of the remote interface) can delegate to the bean. The business methods for the `AgencyBean` have the same signature as those in the remote interface, with the exception that they do not throw `java.rmi.RemoteException`. Those are the steps to specifying a stateless Session bean's interface. Indeed, as you will see later today and tomorrow, specifying the interface of stateful Session beans and of Entity beans follows along very similar lines. In the next section, "Implementing a Stateless Session Bean," you will see the implementation of some of these methods.

Implementing a Stateless Session Bean

Implementing a Session bean involves providing an implementation for the methods of the `javax.ejb.SessionBean`, corresponding methods for each method in the home interface, and a method for each method in the remote interface.

Implementing `javax.ejb.SessionBean`

The implementation of the methods of the SessionBean interface is often boilerplate. The setSessionContext() method usually just saves the supplied SessionContext object:

```
private SessionContext ctx;
public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
}
```

Although `ejbRemove()` method is part of the `SessionBean` interface, you'll learn about its implementation in the next section. As already noted, for a stateless Session bean, the `ejbActivate()` and `ejbPassivate()` methods should have a `null` implementation:

```
public void ejbActivate() { }
public void ejbPassivate() { }
```

Implementing the Home Interface Methods

The home interface has a single method `create()`. The `ejbCreate()` method in the bean corresponds to this method. It makes sense to look up JNDI references in the `ejbCreate()` method and store them in instance variables. This is shown in Listing 5.2.

LISTING 5.2 `AgencyBean.ejbCreate()` Method

```
1: private DataSource dataSource;
2: private String name = "";
3: public void ejbCreate () throws CreateException {
```

LISTING 5.2 Continued

```
 4:     InitialContext ic = null;
 5:     try {
 6:         ic = new InitialContext();
 7:         dataSource = (DataSource)ic.lookup("java:comp/env/jdbc/Agency");
 8:     }
 9:     catch (NamingException ex) {
10:         error("Error connecting to java:comp/env/jdbc/Agency:",ex);
11:         return;
12:     }
13:     try {
14:         name = (String)ic.lookup("java:comp/env/AgencyName");
15:     }
16:     catch (NamingException ex) {
17:         error("Error looking up java:comp/env/AgencyName:",ex);
18:     }
19: }
```

On Day 18, "Patterns," you will learn about a design pattern that simplifies JNDI lookups. It can also speed up your beans; some EJB containers are not particularly efficient at obtaining references from within JNDI.

In this case, the `ejbCreate()` method makes two lookups from JNDI; the first is to obtain a `DataSource` (you will see how this is used shortly) and the other is to obtain environment configuration information—that is, the name of the agency—from the deployment descriptor. You will learn about the deployment descriptor in the following section.

Incidentally, this is a good place to note that stateless Session bean does not mean that the bean has no state; just that it has no state that is specific to any given client. In the case of the `Agency` bean, it caches a `DataSource` and its name in instance variables.

The home interface inherits a `remove(Object o)` from `EJBHome`. This corresponds to the `ejbRemove()` method of the bean. The implementation is pretty simple; it should just reset state:

```
public void ejbRemove(){
    dataSource = null;
}
```

The `ejbRemove()` method is mandated by the EJB specification in three different ways! It appears in the `SessionBean` interface, and it is required as the method corresponding to the home method of `remove(Object)` (inherited from `javax.ejb.EJBHome`) and is also required as the method corresponding to the remote method of `remove()` (inherited from `javax.ejb.EJBObject`). It is covered here because it fits best along side the coverage of `ejbCreate()`.

Implementing the Remote Interface Methods

The remaining methods correspond to the business methods defined in the remote interface. The `Agency` session bean manipulates the data in the `Applicant`, `Customer`, `Location`, and `Skill` tables, providing methods to return all the data in a table, to insert a new item, or to delete an existing item. When deleting rows, rows in dependent tables are also removed.

The methods that manipulate the database all require a `java.sql.Connection` to submit SQL to the database. In regular "fat client" applications, the idiom is to create a database connection at application startup and to close the connection only when the user quits the application. This idiom exists because making database connections is expensive in performance terms. When writing EJBs, however, the idiom is the precise opposite. You should obtain the database connection just before it is needed, and close it as soon as your processing is complete. In other words, "acquire late, release early." This is because the EJB container has already made the database connections and holds them in a pool. When your bean obtains its connection, it is simply being "leased" one from the pool for a period of time. When the connection is "closed," in reality it is simply returned back to the pool to be used again.

The `getLocations()` method shows this principle clearly, as shown in Listing 5.3.

LISTING 5.3 `AgencyBean.getLocations()` Method

```
1: public Collection getLocations() {
2:     Connection con = null;
3:     PreparedStatement stmt = null;
4:     ResultSet rs = null;
5:     try {
6:         con = dataSource.getConnection();
7:         stmt = con.prepareStatement("SELECT name FROM Location");
8:         rs = stmt.executeQuery();
9:
```

LISTING 5.3 Continued

```
10:         Collection col = new TreeSet();
11:         while (rs.next()) {
12:             col.add(rs.getString(1));
13:         }
14:
15:         return col;
16:     }
17:     catch (SQLException e) {
18:         error("Error getting Location list",e);
19:     }
20:     finally {
21:         closeConnection(con, stmt, rs);
22:     }
23:     return null;
24: }
25: private void closeConnection (Connection con,
26:              PreparedStatement stmt, ResultSet rslt) {
27:    if (rslt != null) {
28:       try {
29:           rslt.close();
30:       }
31:       catch (SQLException e) {}
32:   }
33:   if (stmt != null) {
34:       try {
35:           stmt.close();
36:       }
37:       catch (SQLException e) {}
38:   }
39:   if (con != null) {
40:       try {
41:           con.close();
42:       }
43:       catch (SQLException e) {}
44:   }
45: }
```

In this method, you can see the `DataSource` object obtained in the `ejbCreate()` method in use. The `Connection` object is obtained from this `DataSource` object. Another advantage of this approach is that the user and password information does not need to be embedded within the code; rather, it is set up by the deployer who configures the `DataSource` using vendor-specific tools. As you remember from Day 2, "The J2EE Platform and Roles," in the J2EE RI, the `DataSource` object is configured by editing the `resource.properties` file in the `%J2EE_HOME%\config` directory.

The other business methods all access the database in a similar manner.

Exceptions

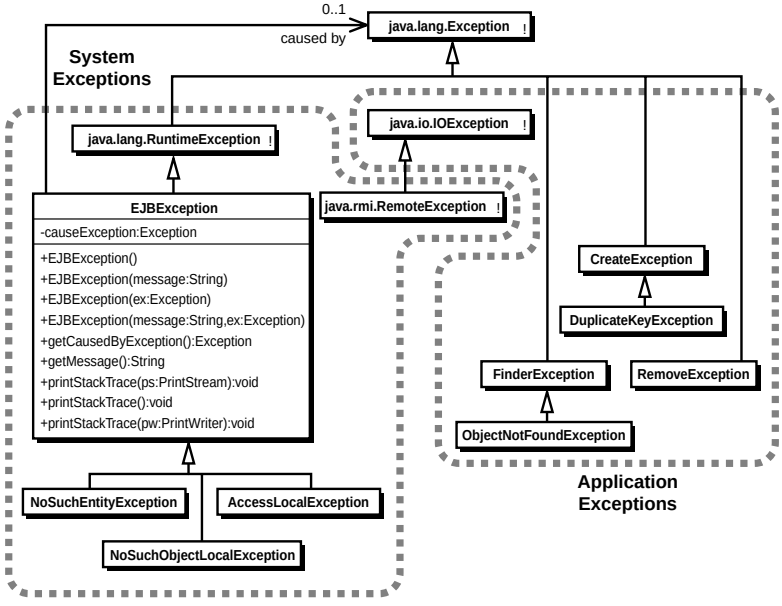
Your bean needs to be able to indicate when it hits an exception. The EJB specification lays out certain rules as to the types of exceptions your bean can throw, because the client does not call your bean directly. For remote clients, there is also the possibility of network problems.

The EJB specification categorizes exceptions as either application exceptions or system exceptions. These correspond quite closely to the regular Java categories of checked exceptions and runtime exceptions.

Figure 5.5 shows the exceptions in the `javax.ejb` package, indicating which are application and which are system exceptions.

FIGURE 5.5

Exceptions are either system exceptions or application exceptions.



So, what do these categorizations mean? If a bean throws an application exception, the EJB container will propagate this back to the application client. As you shall see on Day 8, "Transactions and Persistence," any ongoing transaction is not terminated by an application exception. In other words, the semantics of an application exception are pretty similar to a checked exception; generally, the client can recover if desired.

However, if a bean throws a system exception, that indicates a severe problem that will not be recoverable by the client. For example, if the bean has been incorrectly deployed such that the database connection fails, there is very little that the client can do about it.

In such a case, the EJB container will take steps to terminate any ongoing transaction because it is unlikely to complete. Moreover, the EJB container will discard the bean instance that threw the exception. In other words, there is no need to code any clean up logic in your bean after having thrown a system exception.

Although all runtime exceptions are classified as EJB system exceptions, the `javax.ejb.EJBException` is a `RuntimeException` provided for your use. This class allows the underlying cause to be wrapped through one of its constructors. The `error()` helper method in `AgencyBean` does precisely this:

```
private void error (String msg, Exception ex) {
    String s = "AgencyBean: "+msg + "\n" + ex;
    System.out.println(s);
    throw new EJBException(s,ex);
}
```

In Figure 5.5, you can see that there is one checked exception, namely `java.rmi.RemoteException`, that is classified as an EJB system exception rather than as an EJB application exception. Your bean code should never throw this exception; instead, it is reserved for the EJB container itself to throw. If your bean has hit a system exception, it should throw an `EJBException` rather than `RemoteException`.

Configuring and Deploying a Stateless Session Bean

With the code compiled, the next step is to deploy the bean onto the EJB container.

As you learned yesterday, EJBs are designed to be portable across EJB containers, and the configuration information that defines the bean's name, interfaces, class(es), characteristics, dependencies, and so on is stored in an XML document called a deployment descriptor. This is provided along with the bean code itself.

As you appreciate from Day 2, there are several EJB roles involved in building the deployment descriptor. The bean provider specifies the information about a given bean ("intra-bean" configuration information, if you like), and the application assembler specifies the information about all the beans in an application ("inter-bean" configuration information). When both the bean deployer and application assembler have specified their information, the deployment descriptor is complete.

However, that's not the end of the story, because the deployment descriptor does not define every piece of configuration information necessary to deploy a bean. In effect, the deployment descriptor defines only the *logical* relationships and dependencies between the beans. There will also be additional configuration information that maps the logical

dependencies of the deployment descriptor to the *physical* environment. Performing this mapping is the role of the deployer.

EJB container vendors are free to capture this additional mapping in any way they want, although most use auxiliary deployment descriptors, again usually XML documents. In the case of the J2EE RI, the auxiliary deployment descriptors are indeed XML documents. The EJB specification explicitly disallows vendors from storing their auxiliary mapping information in the standard deployment descriptor itself.

Thus, to port an EJB from one EJB container to another, all that should be required is to recreate an auxiliary deployment descriptor. In other words, the deployer has to redeploy the application, but the bean provider and the application assembler should not have to get involved.

Because manipulating XML documents can be somewhat error prone, most EJB container vendors provide graphical tools to do the work. As you saw yesterday, this is the `deploytool` GUI in the case of the J2EE RI. Unfortunately, many such tools do not distinguish between information that is being saved in the standard deployment descriptor and that which is being saved in the vendor's own auxiliary deployment descriptors. Also, many tools do not explicitly support the EJB architecture's concept of roles, making it possible for a bean provider to start specifying information that might more correctly be decided only by the application assembler or even the deployer. The J2EE RI `deploytool` is guilty on both counts.

Because you may not be using J2EE RI in your own projects, this section presents the task of deployment by looking at both the J2EE RI `deploytool` and also the underlying XML deployment descriptor. Having a firm understanding as to how these relate should make it much easier for you to deploy if you aim to deploy to some other EJB container. It also has to be said that understanding the XML deployment descriptor makes the `deploytool` GUI easier to comprehend.

Using `deploytool`

This section shows how to deploy the Day 5 version of the case study application to the J2EE RI. It's best if you follow along (but if you're on a train, just read the text and make do).

As usual, start up the Cloudscape RDBMS and J2EE RI using two console windows. Then, start up a third console window and start `deploytool`.

By choosing File, Open, load up the `day05\Examples\agency.ear` enterprise application archive. This defines two groups of Session beans—`Agency` and `Advertise`. Their contents have already been configured to contain the appropriate code. Click either of these in the explorer on the left side of the `deploytool` GUI and their contents will be shown on the right side. Note that for both of these beans, some supporting classes (application exception classes) also constitute part of the bean.

To deploy the Session beans, select the Examples item (under the Applications folder in the explorer area on the left side) and choose the Tools, Deploy menu option. Click Next twice and then click Finish.

As you saw yesterday, the deployment descriptor holds configuration information. This is accessible within `deploytool` as follows. Select the `Agency` element from the explorer, and then choose Tools, Descriptor Viewer from the menu. This will display the XML deployment descriptor for all of the beans in that EJB JAR file (in this case, just the one `Agency` bean). Figure 5.6 shows this screen.

FIGURE 5.6

The `deploytool` lets you view the underlying deployment descriptor.

In the following sections, you'll see how this information is structured and built up.

Structural Elements

XML documents provide a mechanism to store data in a hierarchical format, and are similar in style to HTML documents. You shouldn't have too much trouble following the coming discussion, but if you want to do some additional background reading, skip forward to Day 16, "Integrating XML with J2EE," which covers XML documents in more detail.

The format of the EJB deployment descriptor is defined by a document type definition (DTD) file called `ejb-jar_2_0.dtd` in the `%J2EE_HOME%\lib\dtds\` directory. The root of an EJB deployment descriptor is the `ejb-jar` element, whose definition is as follows:

```
<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?,
enterprise-beans, relationships?, assembly-descriptor?, ejb-client-jar?)>
```

This indicates that an `ejb-jar` element may (the `?` sign suffix) contain one each of a `description`, `display-name`, `small-icon`, `large-icon`, `relationships`, `assembly-descriptor` and `ejb-client-jar` elements, and must (no suffix) contain one `enterprise-beans` element.

The `enterprise-beans` element's definition is as follows:

```
<!ELEMENT enterprise-beans (session | entity | message-driven)+>
```

This states that an `enterprise-beans` element consists of one or many (the `+` sign suffix) elements that are either `session`, `entity`, or `message-driven` elements. In other words, the `enterprise-beans` element contains one or more `session`, `entity`, or `message-driven` elements. You will learn about the `session` element shortly.

In Figure 5.6, you can see this structure in the deployment descriptor, and you can also see the same hierarchy in `deploytool`'s explorer pane on the left side of the explorer. The `Advertise` `ejb-jar` element consists of two beans.

Presentational Elements

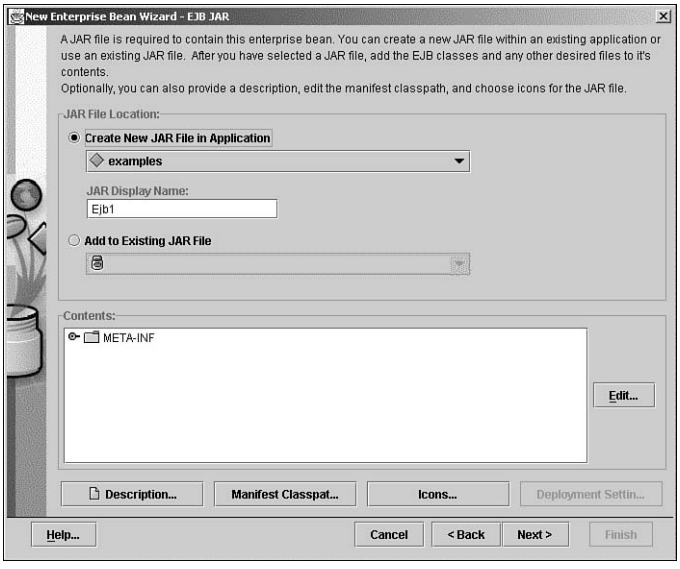
In many of the definitions within the DTD, you will see the `display-name`, `description`, `small-icon`, and `large-icon` elements defined. These are used by vendor deployment tools when managing your beans, so you will certainly want to define a `display-name` to distinguish the beans in the tool's GUI. Whether you choose to provide the remaining elements is up to you. Their presence is primarily so that third- party companies can develop EJBs to sell as "off-the-shelf" business logic components.

In Figure 5.6, you can see that the `display-name` element for the two groups of EJB JARs have been set to `Advertise` and `Agency`. This is shown in the explorer on the left side of `deploytool` GUI. It is also presented as the (read-only) JAR Display Name on the right side when the `Advertise` node is selected. There doesn't appear to be any good reason why this is read-only in `deploytool`, that's just the way the tool works.

If you wanted to add another EJB to the enterprise application, (using the File, New, Enterprise Bean menu option) it would either be in an existing `ejb-jar` or a new `ejb-jar` could be defined. When choosing the second option, the `display-name` for your new collection of EJBs can be specified. This is shown in Figure 5.7.

FIGURE 5.7

The `deploytool` allows EJBs to be defined in either their own `ejb-jar` (with attendant deployment descriptor) or in an existing `ejb-jar`.


New Enterprise Bean Wizard - EJB JAR
A JAR file is required to contain this enterprise bean. You can create a new JAR file within an existing application or use an existing JAR file. After you have selected a JAR file, add the EJB classes and any other desired files to it's contents.
Optionally, you can also provide a description, edit the manifest classpath, and choose icons for the JAR file.
JAR File Location:
Create New JAR File in Application
examples
JAR Display Name:
Ejb1
Add to Existing JAR File
Contents:
META-INF
Edit...
Description...
Manifest Classpat...
Icons...
Deployment Settin...
Help...
Cancel
< Back
Next >
Finish

For your own custom applications, it really is up to you whether you choose to use one `ejb-jar` or several. In the case study example, the latter has been used. Certainly, if you wanted to use some off-the-shelf component EJBs bought from a third-party vendor, the EJBs will already have been bundled into an EJB JAR file. To add these to your enterprise application, you would use File, Add to Application, EJB JAR menu option. When the selected JAR file is read, the `display-name` element from the associated XML deployment descriptor would then be used.

Session Element

The configuration information for Session beans is defined—not surprisingly—in the `session` element of the DTD. Its definition is as follows:

```
<!ELEMENT session (description?, display-name?, small-icon?, large-icon?,
ejb-name, home?, remote?, local-home?, local?, ejb-class,
session-type, transaction-type,
env-entry*,
ejb-ref*, ejb-local-ref*,
security-role-ref*, security-identity?,
resource-ref*,
resource-env-ref*)>
```

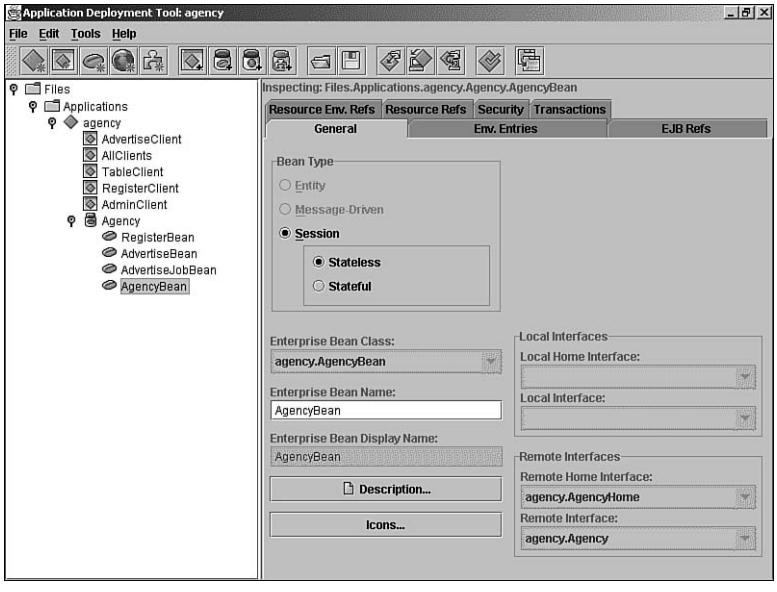
This indicates that a `session` element may contain the presentational elements just discussed, and also will consist of a number of other mandatory and optional elements. The mandatory elements are as follows:

- The `ejb-name` element is the mandatory logical name for the EJB. This must be unique within all the EJBs defined within the `ejb-jar` element.
- The `home`, `remote`, `local-home`, and `local` elements define the remote and local interfaces for the bean. They are all marked as being optional, although the specification also requires that either the `home` and `remote` and/or the `local-home` and `local` elements are defined. As was noted previously, you'll be learning about local interfaces in detail tomorrow.
- The `ejb-class` element defines the class that has the bean's actual implementation.
- The `session-type` element indicates whether the bean is stateless or stateful.
- The `transaction-type` element indicates how the EJB container should manage transactions. You will be learning about this in detail on Day 8, so until then, just specifying that transactions are Required will be sufficient.

This information is available graphically in `deploytool`. Select the `Agency` Session bean in the explorer pane (under `Agency ejb-jar`). You should see something similar to that shown in Figure 5.8.

FIGURE 5.8

The `deploytool` represents the underlying deployment descriptor graphically.



You should be able to see a pretty close resemblance between the information portrayed in `deploytool` and the mandatory information required by the underlying deployment descriptor. The only mandatory item not shown is `transaction-type`; that is on the Transactions tab of the GUI.

The (fragment of the) underlying deployment descriptor for the `Agency` bean that is represented in Figure 5.8 is as follows:

```
<session>
    <display-name>AgencyBean</display-name>
    <ejb-name>AgencyBean</ejb-name>
    <home>agency.AgencyHome</home>
    <remote>agency.Agency</remote>
    <ejb-class>agency.AgencyBean</ejb-class>
    <session-type>Stateless</session-type>
… lines omitted …
</session>
```

This should tie in with the code that was presented earlier today.

Thus far, you have only seen the EJB standard deployment descriptor, but there is also the vendor-specific deployment descriptor for J2EE. The structure of this auxiliary deployment descriptor is defined by `%J2EE_HOME%\lib\dtds\sun-j2ee-ri_1_3.dtd`, but it is not so necessary to learn its structure in detail because it all vendor specific, and `deploytool` allows it to be configured through its GUI.

The auxiliary information in this descriptor maps the logical names to the physical runtime environment. For the Session bean itself, a mapping is required from its logical `ejb-name` to its JNDI name. The following fragment from the auxiliary deployment descriptor shows this:

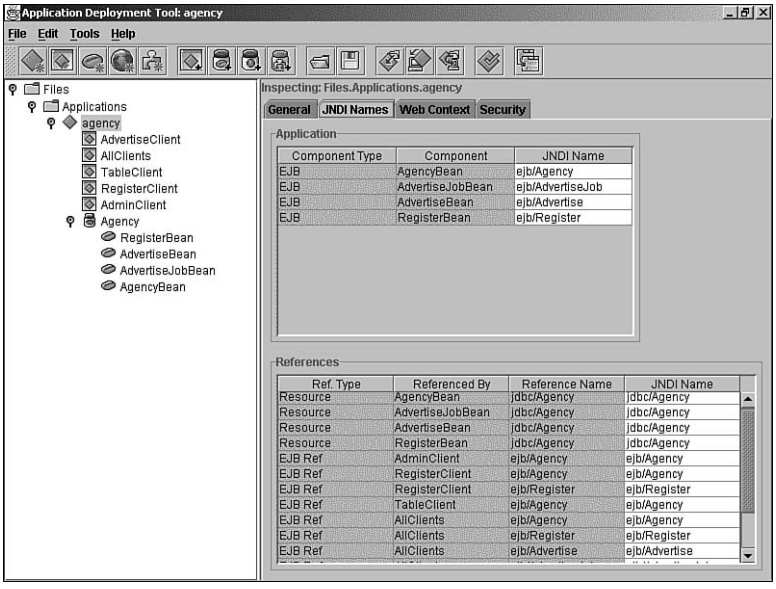
```
<j2ee-ri-specific-information>
    // lines omitted
    <enterprise-beans>
        // code omitted
        <ejb>
            <ejb-name>AgencyBean</ejb-name>
            <jndi-name>ejb/Agency</jndi-name>
```

Figure 5.9 shows how `deploytool` portrays this information. Figure 5.9 also shows the JNDI mappings for references; you will learn about these shortly.

The remaining optional items of the EJB deployment descriptor (`env-entry`, `resource-ref`, and so on) also correspond to the different tabs of the `deploytool` window shown in Figure 5.8. These each indicate different types of dependencies that the bean may have with its runtime environment. The following sections discuss each in turn.

FIGURE 5.9

Behind the scenes, `deploytool` stores the JNDI mappings to an auxiliary vendor-specific deployment descriptor.



Environment Entries

Environment entries allow general configuration information—as might be found in a `.ini` file or in the Windows Registry—to be made available to the bean. Such entries are represented by the `env-entry` element in the deployment description and—not surprisingly—are configured on the Env. Entries tab within `deploytool`.

The `Agency` bean uses an environment entry to look up its name. The relevant code is in the `ejbCreate()` method:

```
InitialContext ic = new InitialContext();
// code omitted
name = (String)ic.lookup("java:comp/env/AgencyName");
```

The EJB specification requires that the EJB container makes the environment entries available under the well-defined context of `java:comp/env`. Therefore, this is needed in the JNDI lookup. However, this prefix is *not* required in the deployment descriptor itself.

The DTD defines the `env-entry` element as follows:

```
<!ELEMENT env-entry (description?, env-entry-name, env-entry-type, env-entry-
value?)>
```

The type of the environment entry (`String`, `Integer`, and so on) is indicated through the `env-entry-type` element. The actual value (`env-entry-value`) is optional, meaning that the bean provider/application assembler does not need to define it. If the actual value isn't specified by these roles, the deployer will need to define the value.

Figure 5.10 shows this information being configured within `deploytool`.

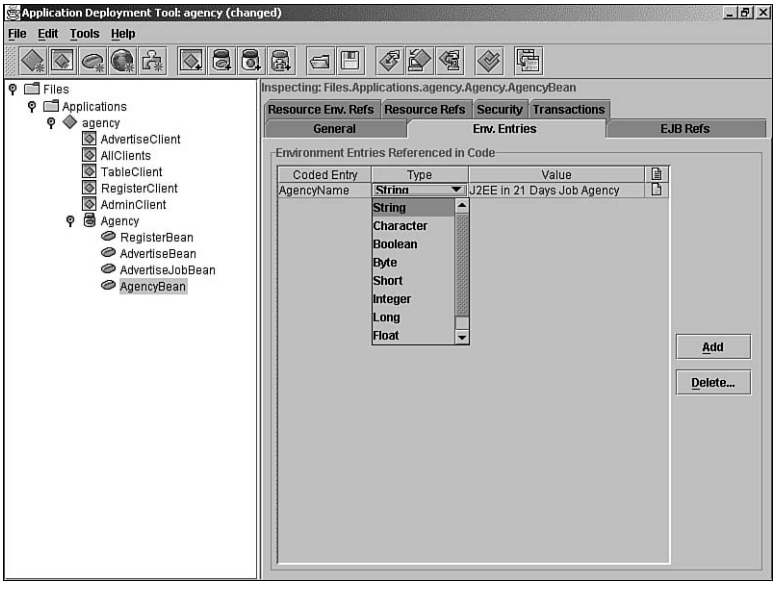


FIGURE 5.10

Environment entries can be managed graphically by `deploytool`.

In the underlying deployment descriptor for the `Agency` bean, this corresponds to

```
<env-entry>
    <env-entry-name>AgencyName</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>J2EE in 21 Days Job Agency</env-entry-value>
</env-entry>
```

To re-emphasize, note that the entry name is `AgencyName`, not `java:comp/env/AgencyName`.

EJB References

Declaring EJB references for an EJB indicates that the bean being deployed depends on other EJBs. Generally speaking, when there are inter-bean dependencies, both the dependent and dependee will have been written by the same bean provider. However, there will be cases when this isn't so. The issue then arises that the dependent EJB may not know the deployed name of the EJB upon which it depends.

For example, a vendor might provide some sort of business-oriented bean (perhaps an `Invoice` EJB) that can optionally perform logging through a `Logging` EJB. The same vendor might well provide an implementation of a `Logging` EJB, but would also allow for application assemblers to use some other EJB that implements the appropriate home and remote interfaces. In this way, the application assembler could ensure that all logging from beans within its application was consistent.

Now the `Invoice` EJB will have a reference to a `Logging` EJB in its JNDI lookup. This might be something like the following:

```
InitialContext ic = new InitialContext();
Object lookup = ic.lookup("ejb/Logger");
LoggerHome home = (AgencyHome)PortableRemoteObject.narrow(lookup,
LoggerHome.class);
Logger logger = home.create();
```

The `"ejb/Logger"` string is hard-coded into the `Invoice` EJB source code and cannot be changed by the application assembler or by the deployer. This is what is sometimes referred to as the *coded name* or *coded entry*. However, the deployment descriptor allows this logical name to be associated with an actual physical name through the `ejb-ref` elements.

The following is the `ejb-ref` element, as defined by the DTD:

```
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote, ejb-
link?)>
```

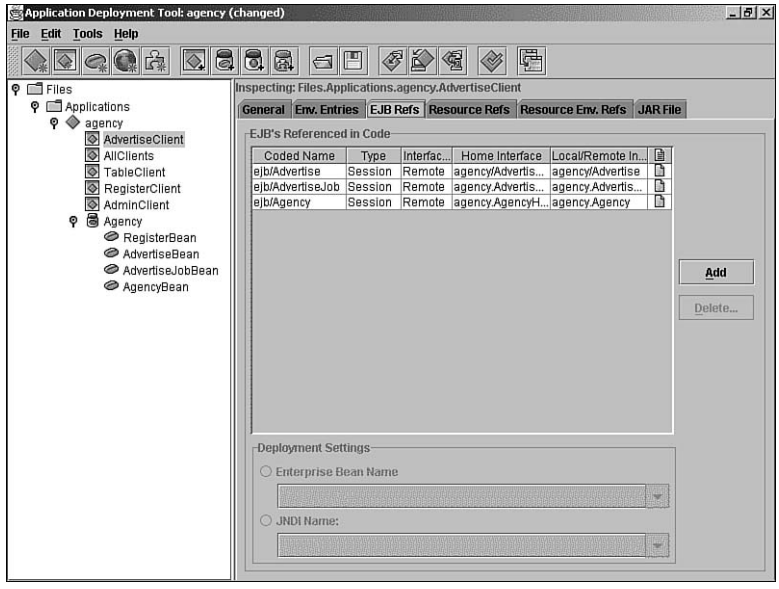
The `ejb-ref-name` element is the coded name—`ejb/Logger` in the previous example. The names of the home and remote interfaces must be specified. Finally, the `ejb-link` element specifies the actual `ejb-name` of the EJB that implements these interfaces. This is a way for the application assembler to constrain the reference in the dependent EJB to a particular EJB within the enterprise application.

In the Day 5 version of the case study, there are no EJB references between EJBs (you will see some such references between EJBs tomorrow), but there are EJB references from the clients and the EJBs. These are shown in Figure 5.11.

When EJB references are defined, they must also be mapped to the physical environment. Figure 5.9 showed the mapping of EJBs and of references to JNDI names. These are shown in the bottom half of the window of Figure 5.9. As an experiment, try temporarily creating a EJB (remote) reference between the beans themselves, and then return to the JNDI tab in `deploytool` (as shown in Figure 5.9). Here, the deployer should indicate the JNDI name for the EJB that implements the remote interfaces. As required by the EJB specification, `deploytool` validates that the JNDI name that is mapped by the deployer to the EJB reference is compatible with any `ejb-link` that may have been specified by the application assembler. When you have finished experimenting, you should delete these EJB references.

FIGURE 5.11

There are EJB references from the clients to the EJBs.



Resource References

Yesterday, you learned that the EJB container allows `DataSources` to be obtained via JNDI. Within the deployment descriptor, a resource reference is used to define that dependency of the EJB. The term *resource reference* is used instead of *database reference* because an EJB can depend on data resources other than databases. It could also depend on an e-mail session (Day 11, "JavaMail"), on a URL, on a message topic or queue factory (Day 10, "Message-Driven Beans"), or on a general resource as defined by the Connector architecture (Day 19, "Integrating with External Resources").

The `Agency` bean has a dependency on a `DataSource` reference that it refers to as `jdbc/Agency`. This can be seen in the `ejbCreate()` method of the `AgencyBean` code. As with the environment entries, note that resource reference has been bound by the EJB container under the context `java:comp/env`:

```
InitialContext ic = new InitialContext();
dataSource = (DataSource)ic.lookup("java:comp/env/jdbc/Agency");
```

Resource references are defined in the DTD as follows:

```
<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth, res-
sharing-scope?)>
```

Going through these in turn

- The `res-ref-name` element is the coded name of the referenced resource.
- The `res-type` element is the fully-qualified interface or class name of the resource.

- The `res-auth` element indicates whether the container will provide the authentication information (username and password) or the application itself. In other words, it indicates which overloaded version of `DataSource.getConnection()` will be called—the one where username and password are supplied (Application authentication) or where they are not (Container authentication).
- The `res-sharing-scope` element indicates whether this resource can be shared among beans (the default) or whether a separate resource will be set up for this bean's exclusive use.

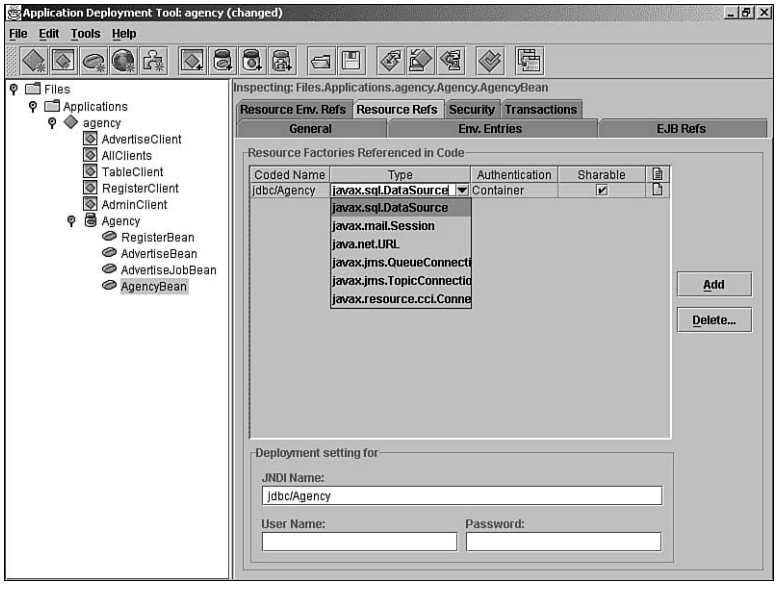
So, in the deployment descriptor for the `Agency` bean, you will see the following:

```
<resource-ref>
    <res-ref-name>jdbc/Agency</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

This information is also shown in `deploytool`, as shown in Figure 5.12.

FIGURE 5.12

Resource references can be managed graphically by `deploytool`.



Again, resource references must be mapped to physical resources. You defined the physical resources for the case study on Day 2, when you edited the `resource.properties` file within `%J2EE_HOME%\config`. The entries that are relevant to this discussion are as follows:

```
jdbcDataSource.5.name=jdbc/Agency
jdbcDataSource.5.url=jdbc:cloudscape:rmi:Agency
```

This instructs J2EE RI to create a `DataSource` using a URL of `jdbc:cloudscape:rmi:Agency` and bind it into JNDI with a name of `jdbc/Agency`.

From the deployment descriptor, you can see that the declared resource reference in the `AgencyBean` Session bean code is `jdbc/Agency` and is also mapped to a JNDI name of `jdbc/Agency`. The fact that the logical and physical names are the same string strings can be a source of confusion; the point is that both are needed. Moreover, the logical resource reference could have been anything at all, so long as it is the same string that appears in the code, in the standard `ejb-jar.xml` deployment descriptor and the vendor-specific auxiliary deployment descriptor.

The mapping between these two names is performed on the JNDI tab of `deploytool`, as shown in Figure 5.9. You should be able to see there that, for example, the resource reference in the `Agency` bean called `jdbc/Agency` maps onto the JNDI name of `jdbc/Agency`. The auxiliary deployment descriptor has the following entries:

```
<j2ee-ri-specific-information>
    // lines omitted
    <enterprise-beans>
        // lines omitted
        <ejb>
            <ejb-name>AgencyBean</ejb-name>
            // lines omitted
            <resource-ref>
                <res-ref-name>jdbc/Agency</res-ref-name>
                <jndi-name>jdbc/Agency</jndi-name>
            </resource-ref>
        </ejb>
        // remaining lines omitted
```

Resource Environment References

Resource environment reference entries allow access to so-called “administered objects.” In J2EE RI, this means JMS queues or JMS topics. In future versions of J2EE, other administered objects may well be specified.

Resource references and resource environment references sound very similar, and, indeed, they are related. However, a resource reference is access to some sort of factory object, used to manufacture (variously) database connections, URLs, JMS sessions, and so on. On the other hand, an administered object must be defined up-front by an administrator and is persistent. The EJB specification doesn’t define RDBMS tables as administered objects, but it might well have. If it had, a resource reference would be to a database connection, and a resource environment reference might be to a table on the database to which you have connected.

The DTD defines resource environment references as follows:

```
<!ELEMENT resource-env-ref (
          description?, resource-env-ref-name, resource-env-ref-type)>
```

The `resource-env-ref-name` is the name of the reference in the code, less the `java:comp/env` prefix, and the `resource-env-ref-type` is either `javax.jms.Queue` or `javax.jms.Topic`.

The `Agency` session bean does not have any dependencies on resource environment references, so there is no screen shot of the `deploytool`. However, you will be using resource environment references on Day 10 when you work with the Java Messaging Service and Message-driven beans. But (again) as an experiment, try adding an resource environment reference on the Resource Env. Refs tab. On the JNDI tab (as shown in Figure 5.9), you should be able to indicate the JNDI name for the administered object. When you have finished experimenting, delete the resource environment reference.

Deploying the Enterprise Application

The enterprise application can be deployed from `deploytool` by using the Tools, Deploy menu option. This causes the bean's code components to be compiled and the proxy and home interfaces to be generated and then packaged up into an `ejb-jar` using information from the underlying deployment descriptor and auxiliary deployment descriptor. Finally, the package is deployed across the network to the J2EE RI.

Once deployed, you can run your application. To do this, use `day05\agency\run\runAll`.

Stateful Session Bean Lifecycle

Now that you have learned how to specify, implement, and deploy stateless Session beans, it is time to look at stateful Session beans. As you shall see, there are many similarities (especially with regard to the deployment), but the lifecycle *is* different and warrants close attention.

The client's view of the lifecycle of a stateful Session bean is identical to that of a stateless Session bean and, in truth, more closely matches the actual lifecycle as managed by the container. Figure 5.13 shows this lifecycle.

From the client's perspective, the bean is simply instantiated when the client calls `create()` on the bean's home interface, and it's removed when the bean calls `remove()` on the bean itself.

The bean's viewpoint of its lifecycle is as shown in Figure 5.14.

The principle difference between stateful and stateless Session beans is the duration of the time that the bean is bound to the client. With a stateless Session bean, the duration was only as long as the time needed to execute the business method. With a stateful Session bean, however, the bean stays bound until the client releases it. In this way, there is a quite close correspondence between the client's and the bean's perspectives.

FIGURE 5.13

The client's view of the lifecycle of stateful beans is identical to that of stateless Session beans.

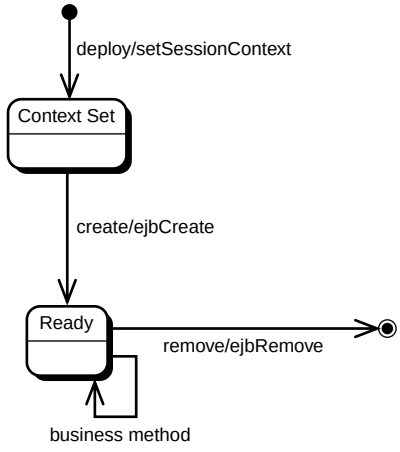
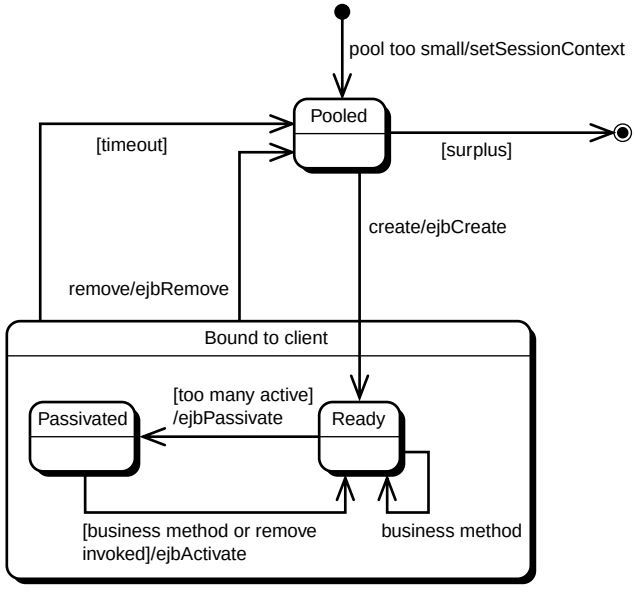


FIGURE 5.14

A stateful Session bean's view of its lifecycle includes passivation and timeouts.



When the client calls `create()` on the home interface, a Session bean instance is obtained. Most EJB containers maintain a pool of unbound Session bean instances, so any unused instance will be chosen. This is then bound to the client. The client can call business methods on the bean, and because the bean will remain bound, these can legitimately save to instance variables the state pertaining to the client. When the client is done with the bean, it calls `remove()` which releases the bean back to the pool of unbound instances.

The EJB specification uses the analogy of a shopping cart, and it is easy to see that this is a natural fit. In such a case, the client would obtain a shopping cart bean using `create()`, call methods such as `addItem()`, `removeItem()`, and `checkout()`, and then release the bean using `remove()`.

If there are many concurrent clients, the amount of memory to manage all of the clients' state can become significant. Moreover, there is nothing to prevent a client from acquiring a bean, and then not using it—an abandoned shopping cart in the aisles, if you like. The EJB specification addresses these issues by defining the notions of passivation and of timeouts. Passivation allows the EJB container to release the memory used by a bean, first writing out its internal state to some secondary storage. This is transparent to the bean's client; when the bean is next used, the EJB container first activates the bean. How the EJB container actually implements passivation is not specified, but the specification does require that the Session bean is serializable, so many implementations will take this route and serialize the bean to a file. If a bean is not used for longer than its timeout, it can be timed out and its memory released.

Perhaps surprisingly, the EJB specification does not define how the timeout for a Session bean reference is specified. Section 7.6.3 of the specification indicates clearly that its definition is specific to the EJB container. Usually, the information will be captured in a vendor deployment tool and stored in an auxiliary deployment descriptor.

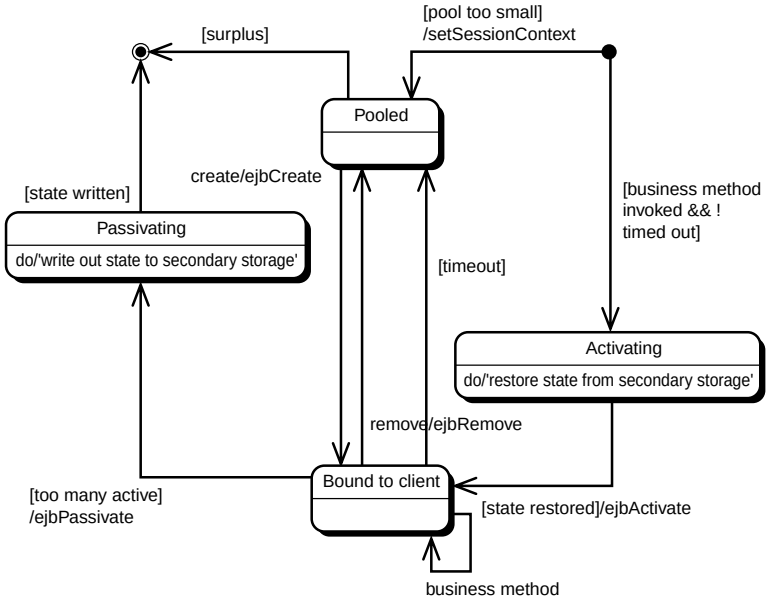
Equally, the EJB specification does not indicate how the EJB container should decide when to passivate beans (though it does suggest that a "least recently used" strategy can be employed, see section 7.6).

Figure 5.14 showed the bean's viewpoint of its lifecycle, but the actual lifecycle as managed by the EJB container is likely to be different again. After all, the whole point of passivation is to reduce the number of bean instances; if the bean was merely "put to sleep," that wouldn't have been accomplished, so Figure 5.15 shows the actual lifecycle used by many EJB container implementations.

When the EJB container passivates a bean, its state is written out to secondary storage. The bean instance is then destroyed. If a client whose bean instance has been passivated invokes a method, the EJB container first re-instantiates the bean by deserializing it from secondary storage. The business method is then invoked.

FIGURE 5.15

The actual lifecycle of stateful Session beans as managed by the EJB container is somewhat more complex.



Specifying a Stateful Session Bean

Specifying a stateful Session bean is similar to specifying a stateless Session bean. The remote interface defines access to the bean by remote clients, and every method of the remote interface must throw an `RemoteException`. The primary difference (from a specification viewpoint) is that there can be multiple `create()` methods in the home interface.

You will recall that a stateless Session bean allows only for a single `create()` method in the home interface, and this corresponds to the `ejbCreate()` method of the bean itself. For a stateful Session bean, the `create()` method can be overloaded, so that the stateful bean can be given some initial state. From the client's viewpoint, this is somewhat analogous to invoking a constructor on the bean.

For example, the `Advertise` bean in the case study is stateful. It represents an advertiser of job positions. The home interface for this bean is as follows:

```
package agency;

import java.rmi.*;
import javax.ejb.*;

public interface AdvertiseHome extends EJBHome
```

```
{
   Advertise create (String login) throws RemoteException, CreateException;
}
```

Obviously, the `create()` method has a corresponding `ejbCreate()` method in the `AdvertiseBean` class itself. This `ejbCreate()` method must instantiate the bean with any appropriate state, as shown in listing 5.4.

LISTING 5.4 `AdvertiseBean.ejbCreate()` Method

```
 1: package agency;
 2: // imports omitted
 3:
 4: public class AdvertiseBean implements SessionBean
 5: {
 6:     private String login;
 7:     private String name;
 8:     private String email;
 9:     private String[] address;
10:     private Collection jobs = new TreeSet();
11:
12:     public void ejbCreate (String login) throws CreateException {
13:         this.login = login;
14:
15:         // database detail not shown
16:         name = …;
17:         email = …;
18:         address[0] = …;
19:         address[1] = …;
20:         jobs = new TreeSet();
21:         while(…) {
22:             jobs.add( … );
23:         }
24:      }
25: }
```

Alternatively, the EJB specification allows for methods named `create`*`XXX`*`()` to be defined in the home interface, with corresponding methods `ejbCreate`*`XXX`*`()`. These methods can take parameters if required. Whether you choose to use this facility or just use regular overloaded versions of `create()`/`ejbCreate()` is up to you.

Other than this one change of being able to pass in state in the `create()` method, there really is little difference in the specification of a stateful bean compared to that of a stateless Session bean. The remote interface of the stateful `Advertise` Session bean is shown in Listing 5.5.

LISTING 5.5 The Remote Interface for the Stateful `Advertise` Bean

```
 1: package agency;
 2:
 3: import java.rmi.*;
 4: import javax.ejb.*;
 5:
 6: public interface Advertise extends EJBObject
 7: {
 8:     void updateDetails (String name, String email, String[] Address)
          ➥throws RemoteException;
 9:     String getName() throws RemoteException;
10:     String getEmail() throws RemoteException;
11:     String[] getAddress() throws RemoteException;
12:     String[] getJobs() throws RemoteException;
13:
14:     void createJob (String ref) throws RemoteException,
          ➥DuplicateException, CreateException;
15:     void deleteJob (String ref) throws RemoteException, NotFoundException;
16: }
```

The `ejbCreate()` method from the home interface supplies the information to the bean so that it can retrieve the data about the advertiser from the database. The remote interface allows this information to be accessed and be updated.

Implementing a Stateful Session Bean

When implementing a stateful Session bean, there are a number of issues that you must keep in mind. They are discussed in this section.

Passivation

Unlike stateless Session beans, stateful Session beans are at liberty to store client-specific state information in instance variables. However, because your bean may be passivated, any instance variables you define must be either primitives, references to serializable objects, or—failing that—be `transient`.

Of course, any `transient` variables will be reset to `null` if the bean is passivated and then re-activated, so your implementation will need to deal with this. Classes that are not serializable often depend in some way on the environment, such as an open `java.net.Socket`, so that your bean can act as a network client to some service. The general approach to dealing with this is to store other data that *is* serializable in instance variables during the `ejbPassivate()` method. Then, the non-serializable reference can be re-instantiated in the `ejbActivate()` method using this other data.

For example, in the case of a `Socket`, your bean could hold a `String` and an `int` representing the hostname and the port number of the socket. The instance variables and `ejbActivate()` method could would be something like the following:

```
import java.net.*;
// code omitted.

private transient Socket clientSocket;
private String socketHost;
private int socketPort;
public void ejbActivate() {
    this.clientSocket = new Socket(socketHost, socketPort);
}
```

Although your Session bean must itself be serializable, it is not necessary to explicitly implement the `java.io.Serializable` interface. This is because the `javax.ejb.SessionBean` interface extends from `Serializable` (by way of its superinterface, `javax.ejb.EnterpriseBean`).

Given that passivation causes quite a few implementation headaches, some commentators have asked why the EJB specification goes to such lengths to define a passivation mechanism. After all, operating systems are very good at paging memory to disk, and this is all that the "secondary storage" is really accomplishing.

These are good questions, with no ready answers. But if you would rather have the operating system do the work and keep your beans simple, just configure your beans with a very high passivation threshold.

You may be wondering how to handle passivation in a stateful Session bean that has a reference to a home or remote interface of another EJB, a `javax.sql.DataSource` or some other resource connection factory. After all, none of these references may be serializable. Luckily, though, the EJB specification puts the onus for worrying about these references on the EJB container (section 7.4.1). In other words, your bean is at liberty to hold references to any of these types of objects, and you don't need to care whether they are serializable. Of course, most EJB container vendors are likely to comply with this part of the specification by making sure that these objects *are* serializable.

Timeouts

Another difference between stateful and stateless Session beans is that stateful Session beans may timeout. If a bean is timed out, the client's reference is no longer valid, and any further invocations on that reference will result in a `java.rmi.NoSuchObjectException` for remote clients.

You should note from Figure 5.14 that if a bean times out, its `ejbRemove()` method will not be called. This means that you shouldn't adopt a convention of acquiring external resources in `ejbCreate()` with a view to releasing them in `ejbRemove()`. Even releasing the resources in `ejbPassivate()` is not enough, because a bean can be timed out even from its ready state.

Don't confuse passivation and timeout. An EJB container might implement passivation using an LRU strategy and allow a bean timeout to be specified in seconds. If the EJB container is not busy, a bean will not be passivated according to the LRU strategy, but it may hit its timeout nevertheless.

Chaining State

It is generally a bad idea to have more than one stateful Session bean involved in any conversation, because no matter which way you cut it, there's always the chance that one of them will time out, preventing the other from completing.

To see this, suppose the client calls stateful Session A, which, in turn, uses the services of stateful Session B. There are two cases—the timeout of A is larger than that of B, or the timeout is less than that of B. Taking each in turn

- Suppose that the timeout of Session bean A is 30 minutes and that of Session bean B is 20 minutes. The client makes a call on A at time t_1=0, which then calls B. If the client calls A at time $t_2 = t_1$ + 25 minutes, A's call to B will fail because B will have timed out.
- Suppose now that the timeout of Session bean A is 20 minutes, and that of Session bean B is 30 minutes. The client makes a call on A at time t_1=0, which then calls B. The client then calls A again at time $t_2 = t_1$ + 19 minutes, although for this call, A does not need to call B to service the request. If the client calls A again at time $t_3 = t_2$ + 19 minutes = t_1 + 38 minutes, A's call to B will fail because B was last invoked more than 30 minutes ago and will have timed out.

A similar problem can occur with session beans and servlets; this is discussed on Day 12.

Configuring and Deploying a Stateful Session Bean

Configuring and deploying a stateful Session bean is just the same as deploying a stateless Session bean. The only difference is that the `session-type` element in the deployment descriptor will be set to `Stateful`.

Client's View

Yesterday, you saw how to use JNDI to obtain a reference to a Session bean home and how to obtain a Session bean by calling the appropriate `create()` method. Now that you have a full understanding of how Session beans work, there are a few other points that are worth appreciating.

First, if your client has a reference to a stateless Session bean, although it should call `remove()` when it is finished with the EJB, this method call doesn't actually do particularly much. In particular, it won't release any bean resources itself, as shown clearly by the state chart diagrams in Figure 5.3. What this will do is allow the EJB container to remove the `EJBObject` proxy for the bean.

Conversely, calling `create()` for a stateless Session bean doesn't necessarily cause `ejbCreate()` to be called on the underlying bean, although the client will have a reference to an `EJBObject` after making this call.

One benefit of stateless beans over stateful is that they are more resilient. That is, if the client invokes a method on a stateless bean and it throws an exception, the client can still use their reference to try again. The client does not need to discard the reference and obtain a new one from the home interface. This is because, behind the scene, the EJB container will have discarded the bean that threw the exception, but can simply select another bean from the pool to service the client's retry attempt. This is safe to do because the stateless Session beans have no state. Of course, it is possible that the retry attempt might fail; it would depend on the underlying cause of the exception.

In contrast, if a stateful Session bean throws an exception, the client must obtain a new Session bean reference and start its conversation over again. This is because the EJB container will have discarded the Session bean that threw the exception, discarding all the client's conversational state in the process.

Sometimes, a client may end up having two references to a Session bean. It may have obtained them both from other method calls, for example. More precisely, the client will have two RMI stubs to Session beans. If the client wants to determine whether these two stubs refer to the same Session bean, it should not call the `equals()` method. This almost certainly will not return `true`. Instead, the client should call `isIdentical(EJBObject)` on the reference. This indicates whether both stubs refer to the same Session bean. Note that two references to the same stateless Session bean will always return `true`, because—at least conceptually—there is only a single instance (see EJB specification, section 7.5.6).

Earlier today, you saw the different types of exceptions that a bean can throw. If a bean throws an application exception, the EJB container will propagate it back to the client. If the bean throws an `EJBException` (representing a system exception), the EJB container will throw a `java.rmi.RemoteException` in turn.

For the client, any `RemoteException` represents a severe problem. It doesn't really matter to the client if the `RemoteException` has arisen because of a network problem or because of a problem with a bean. Either way, it will be unable to recover the problem.

Table 5.1 lists the system exceptions shown in Figure 5.5 and indicates how each is raised and thrown. As you will see, the majority are raised when the EJB container itself has detected a problem with either transactions or security. You will learn more about transactions on Day 8, and more about security a week later on Day 15, "Security."

TABLE 5.1 System Exceptions Are Thrown in a Variety of Situations

What	*Event*	*Client Receives*
Any bean	Throws `javax.ejb.EJBException` (or any subclass)	`java.rmi.RemoteException`
BMP Entity bean	Throws `NoSuchEntityException`	`java.rmi.NoSuchObjectException`
Container	When client invokes method on a reference to a Session bean that no longer exists	`java.rmi.NoSuchObjectException`
	When client calls a method without a transaction context	`javax.transaction.TransactionRequiredException`
	When client has insufficient security access	`java.rmi.AccessException`
	When transaction needs to be rolled back	`javax.transaction.TransactionRolledBackException`

If you are wondering what BMP Entity beans are, the phrase is an abbreviation of "bean-managed persistence Entity beans." You'll be learning about those tomorrow.

Patterns and Idioms

You now know all the important theory behind writing Session beans, but it's always helpful to have one or two real-world insights into how to write them in practice. Patterns (or more fully, "design patterns") are documented pearls of wisdom. Idioms are much the same thing, although they tend to be more lower-level and code-oriented.

On Day 18, many of the design patterns discussed piecemeal throughout the book will be brought together to see how they apply to all aspects to the J2EE environment. Some of

those given here will be presented more fully. But for now, there are patterns and idioms specific to writing session EJBs. Reading this section might save you some time.

Business Interface

One of the peculiarities of the EJB architecture is that there is no direct link between the defined remote interface and the bean that provides the implementation. For example, the remote interface `Advertise` is not implemented by `AdvertiseBean`. However, no link is needed because the EJB specification declares that it is the vendor's deployment tools that are responsible for ensuring that every method defined in the remote interface has a corresponding method in the bean, and that the required methods for the home interfaces also exist.

However, this means that any mismatches between the interface and the bean's implementation will be picked up not during compilation, but during deployment. From a practical point of view, this can make debugging the problem harder. After all, you are probably accomplished at reading compile errors and figuring out what the cause of the problem is. But you won't (at least initially) be familiar with the errors that the vendor's deployment tool throws up when it announces that your bean does not comply with the EJB specification.

One idiom that solves this is to create an additional interface that defines just the business methods. This interface is sometimes called the business interface. Then, the bean implements the business interface, while the remote interface for the bean extends that business interface.

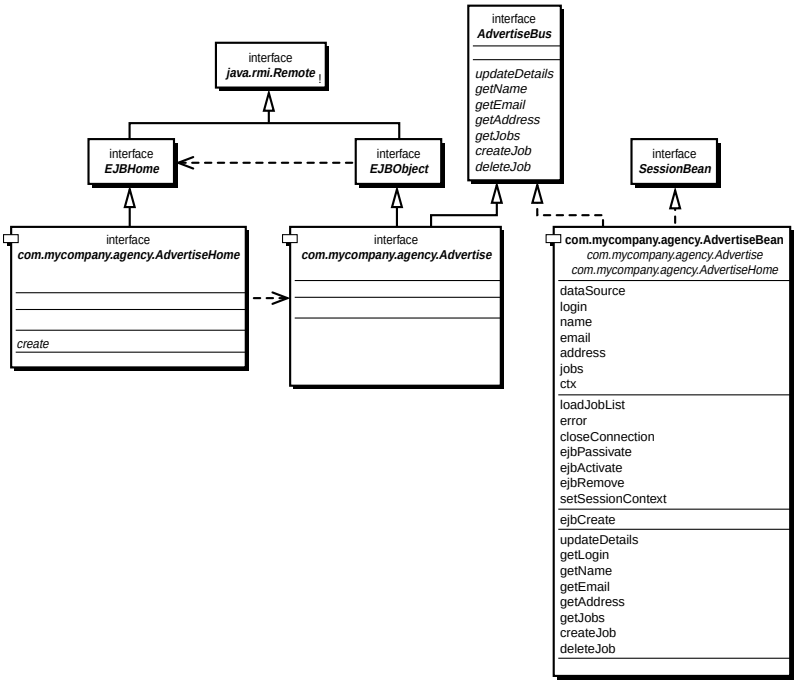
This hasn't been done in the case study, so as not to complicate and confuse. However, it would be simple enough to introduce a business interface. Figure 5.16 shows a UML class diagram that illustrates this for the `Advertise` bean.

With this idiom, if there is a mismatch between the interface and the bean, it will be picked up during compile time.

There is just one subtlety of which you must be aware. When applying this technique to the remote interface of a bean, the methods in the business interface must all throw `java.rmi.RemoteException`. This is because the vendor-generated `EJBObject` for the remote interface must follow the rules of remote objects, so that every one of its public methods throws the `RemoteException`. This applies also to the inherited methods of the business interface. The `AdvertiseBus` interface is shown in Listing 5.6.

FIGURE 5.16

Defining a business interface means that the bean can implement that interface.



LISTING 5.6 `AdvertiseBus` Interface

```
 1: package agency;
 2: import javax.ejb.*;
 3: import java.rmi.RemoteException;
 4:
 5: public interface AdvertiseBus {
 6:     void updateDetails (String name, String email, String[] address)
           ➥ throws RemoteException;
 7:     String getName() throws RemoteException;
 8:     String getEmail() throws RemoteException;
 9:     String[] getAddress() throws RemoteException;
10:     String[] getJobs() throws RemoteException;
11:     void createJob (String ref) throws RemoteException,
            ➥ DuplicateException, CreateException;
12:     void deleteJob (String ref)
            ➥throws java.rmi.RemoteException, NotFoundException;
13: }
```

Adapter

As you write your EJBs, you will quickly find yourself writing reams of "boilerplate" code. For example, the `setSessionContext()` method almost always just saves the

session context to an instance variable. The `ejbActivate()` and `ejbPassivate()` methods often do nothing at all.

If you have written any GUI applications using AWT or Swing, you almost certainly will have used the various `Adapter` classes in the `java.awt.event` package. For example, the `java.awt.event.WindowAdapter` class provides empty implementations of the seven methods in the `java.awt.event.WindowListener` interface.

Adapter classes can also provide common default functionality. For example, the `AbstractList` class acts as an adapter to the `List` interface in the `java.util` package, providing the majority of the implementation required. Although the `List` interface defines 25 methods in total, the `AbstractList` class implements all but two of them.

Creating an adapter for your Session beans can save you time in the long run. You can provide default implementations for many of the lifecycle methods, and can also provide additional methods. For example, you might decide to provide a `log()` method that will forward any log messages to some remote URL or to a logging database.

Coarse-Grained

Remote Session beans should offer coarse-grained services. In other words, the services offered by a remote Session bean should do large(-ish) chunks of work. The overhead of the network to use these beans then becomes much less significant.

There are a number of approaches for arranging this. One approach is to create value object classes. These are serializable objects that encapsulate enough information for the Session bean to provide some service. The client populates these value objects and then sends them across the wire as part of the remote method call. The Session bean then interrogates its copy of the value object to accomplish its work. You will learn about the value object pattern and some related patterns more fully on Day 18.

The value object idea as described is to encapsulate enough data in an object such that the Session bean can do a reasonable chunk of work, but the responsibility for figuring out what that chunk of work is still resides with the Session bean. A natural extension to this concept is to place that responsibility into the value object itself. In effect, the value object represents the action or command to be invoked. Indeed, the name of this design pattern is the Command design pattern.

Gotchas

As you start to implement your own Session beans, there's bound to be a couple of aspects that will trip you up. The following quick checklist of such "gotchas" should keep you on the straight-and-narrow:

- When you look up resources from JNDI, you should use a string of the form `java:comp/env/XXX`. However, in the deployment descriptor, only the *XXX* is needed; the `java:comp/env` prefix is implicit.
- Perhaps obvious, but don't use `ejb` as a prefix for naming your business methods. Names of that format are reserved for the EJB architecture callback methods.
- Don't implement the remote interface in your bean! If you do so, your bean could inadvertently return itself (Java keyword `this`) as a return type. If a client starts invoking methods on this reference, it will bypass all of the EJB container's transaction and security control that is managed within the `EJBObject` proxy to the bean. Instead, use the business interface idiom mentioned earlier today.
- The `EJBObject` interface defines a `getPrimaryKey()` method; the `EJBHome` interface defines a `remove(Object primaryKey)` method. Attempting to call either of these for a Session bean will immediately throw a `RemoteException`, so don't do it. They are there only for Entity beans, discussed tomorrow.
- You'll learn more about transactions on Day 8, but for now, remember that you should not perform work that updates a database in the `ejbCreate` or `ejbRemove` method, or indeed the other `ejb`*XXX*`()` lifecycle methods. This is because the transaction context is undefined. See section 7.5.7 of the EJB specification for more details.
- Don't try to have a Session bean call itself through its own `EJBObject`; it won't work. This is prevented so that the bean developer does not need to worry about multiple threads. In other words, Session beans are non-reentrant. Of course, your bean can call methods on itself directly through its implicit `this` reference.
- An `ejbCreate()` is required for stateless Session beans. It isn't in the `javax.ejb.SessionBean` interface because stateful Session beans won't necessarily have a no-arg `create()` method.

Summary

You've covered a lot of ground today. You've learned that there are stateless and stateful Session beans, and each has their own lifecycle. You've seen in detail how to specify a Session bean by defining its home and remote interfaces and how to implement a bean by providing corresponding implementations for the methods in the home and remote interfaces, as well as how to implement the lifecycle methods as defined in the `javax.ejb.SessionBean` interface.

You've also learned in detail how the deployment descriptor provides configuration information describing the bean's characteristics and dependencies to the EJB container.

Additionally, you've seen that those dependencies are logical dependencies that must be mapped by the EJB deployer role to the physical resources defined through vendor-specific auxiliary deployment descriptor.

Finally, you've learned about some common techniques, design patterns, and idioms that can simplify your coding and that represent best practice.

Q&A

Q What sort of state can stateless Session beans have?

A Somewhat surprisingly, stateless Session beans can store state, but it must be independent of the client.

Q What is the prefix that will appear in all JNDI lookups?

A The `java:comp/env` context is guaranteed to exist in an J2EE environment.

Q How are EJB system exceptions different from regular Java exceptions?

A `RemoteExceptions` can be caused by network problems, which, in the context of distributed J2EE enterprise applications, represent a system-level rather than application-level exception.

Q How is the timeout for a stateful Session bean defined?

A Surprisingly, the mechanism for specifying the timeout interval for a stateful Session bean is not mandated in the EJB specification.

Exercises

The exercise starts with a version of Day 5's job agency case study that already provides a number of beans:

- There is a stateless `Agency` bean that returns lists of all applications, customers, locations and skills in the database.
- There is a stateful `Advertise` bean that allows advertisers (of jobs) to update their name, e-mail, and address, and to manage the jobs they have posted to the job agency.
- There is a stateful `AdvertiseJob` bean that represents an advertised job. This allows the description, location, and required skills to be maintained.

However, it does not define any bean for the potential job applicants at this point. What is required is a `Register` bean that allows applicants to register themselves with the job agency. The exercise is to implement the `RegisterBean`, define this new bean within the supplied `agency.ear` enterprise application, configure the bean, deploy your bean to the J2EE RI, and finally test with either `RegisterClient` or `AllClients` (supplied).

Under the `day05\exercise` directory, you will find a number of subdirectories, including the following:

- `src` The source code for the EJBs and clients.
- `classes` Directory to hold the compiled classes; empty.
- `build` Batch scripts (for Windows and Unix) to compile the source into the classes directory. The scripts are named `compile`*`XXX`*.
- `jar` Holds `agency.ear`—the agency enterprise application. Also holds `agencyClient.jar`, the client-side JAR file optionally generated when deploying EAR. This directory also holds some intermediary JAR files that are used only to create the previous two JAR files.
- `run` Batch scripts (for Windows and Unix) to run the JARs. Use the files in the `jar` directory.

The `Register` and `RegisterHome` interfaces have been provided for you, under the `src` directory. For example, the `Register` interface is as follows:

```
package agency;

import java.rmi.*;
import javax.ejb.*;

public interface Register extends EJBObject
{
void updateDetails (String name, String email,
         ➥         String location, String summary, String[] skills)
         ➥         throws RemoteException;
    String getLogin() throws RemoteException;
    String getName() throws RemoteException;
    String getEmail() throws RemoteException;
    String getLocation() throws RemoteException;
    String getSummary() throws RemoteException;
    String[] getSkills() throws RemoteException;
}
```

Today's exercise is to implement the `RegisterBean`, configure an appropriate deployment descriptor, deploy your bean to the J2EE RI, and then test with the `RegisterClient`. The bean will need to be stateful.

If you need some pointers as to how to go about this, read on.

1. Create a `RegisterBean.java` file and place this in `day05\exercise\src\agency`.
2. Implement `RegisterBean` to support the `Register` and `RegisterHome` interfaces supplied. Base your implementation on that of `AdvertiseBean`, if you want.

3. Compile the `RegisterBean` code and the other interfaces, using the `build\compileAgencySessionEjbs` script. Note that this expects the `JAVA_HOME` and `J2EE_HOME` environment variables to be set.
4. In `deploytool`, open up the existing enterprise application (`day05\exercise\jar\agency.ear`). Then, add the your `Register` bean to the existing `Agency ejb-jar` by using File, New, Enterprise Bean. Specify the contents to include all the required class files.
5. Configure the deployment descriptor for the `RegisterBean` appropriately. The bean will need to be stateful. You will need to specify resource references and JNDI names for the `RegisterBean`; bind the bean to a name of `ejb/Register`.
6. For the `RegisterClient` application client, configure the EJB reference appropriately. This has a coded name of `java:comp/env/ejb/Register` to refer to the `RegisterBean`.
7. Deploy your bean by selecting it and using Tools, Deploy. As you do this, you will need to define the appropriate JNDI mappings (mapping the logical EJB references to the physical runtime environment). Request a client JAR file to be created, called `agencyClient.jar`, to reside in the JAR directory.
8. To test out your bean, compile `AllClients` using the `build\buildAllClientsClient` script. Then run the client using `run\runAll`.

You may also have noticed that in the `build` directory there are several other scripts apart from those to compile the source. In fact, these can be used to recreate the `agency.ear` file using the deployment descriptors held in the `dd` directory. You will be learning more about this approach tomorrow. For now, all that you need to know is that the `agency.ear` file can be created automatically by running the `bat\buildall` script. It requires that the `RegisterBean` class exist to run successfully. You can then use `deploytool` to manually define and configure the `RegisterBean` within the EAR.

The solution to the exercise is under `day05\agency`.

WEEK 1

DAY 6

Entity EJBs

Yesterday, you learned about Session beans, and how they provide a service to a specific client.

The major topics that you will be covering today are

- How Entity beans represent domain objects, providing services that can be used by all clients
- Two types of Entity beans—bean-managed persistence (BMP) and container-managed persistence (CMP)
- How EJBs can provide a local interface in addition to their remote interface
- Specifying, implementing, configuring, and deploying BMP Entity beans
- Configuring and deploying EJBs from the command line rather than using a GUI

Overview

When building IT systems, the functionality required of the application must be specified and the business objects within the domain must be identified.

In “traditional” client/server systems, the application’s functionality can be implemented in the front-end application or perhaps using database stored procedures, and the domain objects are usually tables within an RDBMS. In building an EJB-based system, the application’s functionality corresponds to Session beans, and the domain objects correspond to Entity beans.

You learned yesterday that Session beans take on the responsibility of implementing the application’s business functionality. There will still be a presentation layer to display the state of those Session beans, but its detail is unimportant in the larger scheme of things.

In the same way, Entity beans take on the responsibility of representing the domain data. There will still a persistent data store to manage the data, almost certainly an RDBMS, but the Entity beans abstract out and hide the detail of the persistence mechanism.

The N-tier Architecture Revisited

On the very first day, you were introduced to n-tier architectures, with the business logic residing in its own tier. With an EJB-based system, both Session and Entity beans are objects, so the business logic could be reside in either of them. In practice, the business logic will be split over both, but to make the correct decision, it is worthwhile analyzing what is meant by that phrase “business logic.”

Business logic refers to the collection of rules, constraints, procedures and practices put in place by the business users to conduct their business. Some of the rules and constraints cannot be changed by the business, due to the domain in which the business is performed. For example, there could be legal constraints and obligations. The procedures and practices represent the (one particular) way in which business users have chosen to conduct business.

Rules and constraints generally apply across all applications. In other words, it doesn’t matter what the business is trying to accomplish, they will still need to comply with such rules and constraints. This sort of business logic is best implemented through Entity beans, because Entity beans are domain objects that can be reused in many different applications.

In the business world, procedures and practices are usually the expression of some sort of application, so Session beans are the best vehicle to implement this type of business logic. Indeed, introducing computerized systems often changes these procedures and practices (hopefully for the better, sometimes for the worse) because computers make available new ways of accomplishing tasks.

- Session beans should have the business logic of a specific application—in other words, application logic. The functionality provided should allow the user to accomplish some goal.

- Entity beans represent domain objects and should have business logic that is applicable for all applications—in other words, domain logic. Usually, this logic will be expressed in terms of rules and constraints.

If there is any doubt as to where the functionality should be placed, it is safer to place it with the Session bean. It can always be moved later if it is found to be truly re-usable across applications.

Figure 6.1 shows a UML component diagram to illustrate that there are at least four logical layers in an EJB-based system. Normally, at least some of these layers will be on the same physical tier.

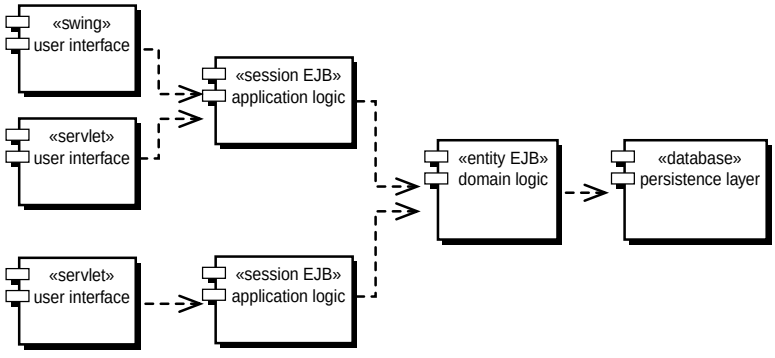


FIGURE 6.1

EJBs separate out business logic into application and domain logic.

Comparison with RDBMS Technology

It's natural to compare Entity beans with relational databases, because there is a significant overlap in the objectives of both technologies.

If you like to think in client/server terms, you could think of Session beans as being an extension of the "client", and Entity beans as being an extension of the "server". It's important to realize that many clients can share a given Entity bean instance at the same time, just as many database clients can read some row from a database table at the same time.

You can also think of Entity beans as a high-performance data cache. Most RDBMS' store data pages or blocks in a cache so that the most commonly used rows in tables can be read directly from memory rather than from disk. Although the EJB specification does not require it, many EJB containers adopt a strategy such that Entity beans are also cached, so the data that they represent can also be read directly from memory. The advantage of the Entity bean cache over an RDBMS' data cache is that the Entity beans already have semantic meaning and can be used directly. In contrast, data read from an RDBMS' data cache needs to be reconstituted in some way before it can be used.

Identifying Entities

At their simplest, Entity beans can correspond to nothing more complex than a row in a database; any data that might reasonably be expected to exist in a relational database table is a candidate. This makes examples of Entity beans easy to come by:

- A `Customer` Entity bean would correspond to a row in a `customer` table keyed by `customer_num`. The list of contact phone numbers for that `Customer` (in a `customer_phone_number` detail table keyed on (`customer_num`, `phone_num`) would also be part of the `Customer` Entity bean.
- An `Invoice` Entity bean might correspond to data in the `order` and `order_detail` tables.
- An `Employee` Entity bean could be persisted in an `employee` table. The employee's salary history might also be part of the Entity bean.

Identifying entities can be made easier if a proper discipline is adopted with relational modeling of the database. Of course, many databases just evolve over time as developers add tables to support new requirements. Ideally, though, there should be a logical database model and a physical database model. The former is usually captured as an Entity relationship diagram (ERD) with entities, attributes, and relationships. Relational database theory defines a process called *normalization* and different *normal forms* that aim to eliminate data redundancy. It is this stage at which the normalization rules are applied, to get to third normal form (at least).

This isn't a book on relational database design, but here's a cute phrase that you can use to get you to third normal form: "every non-key attribute depends upon the key, the whole key, and nothing but the key (so help me Codd!)." If you are wondering who Codd is, that's Dr. Codd who in the early 1970s laid down the mathematical foundations for relational theory.

Converting a logical database model to a physical model is in many ways mechanical. Every entity becomes a table, every attribute becomes a column, and every relationship is expressed through a foreign key column in the "child" table.

These entities identified in logical data modeling are the very same concepts that should be expressed as Entity beans. Moreover, one of the key "deliverables" from performing relational analysis is the selection of the primary key—the attribute or attributes that uniquely identify an instance. Entity beans also require a primary key to be defined, and this is manifested either as an existing class (such as `java.lang.String` or `java.lang.Integer`) or a custom-written class for those cases where the key is composite.

The name often given to such primary key classes is something like `BeanPK`, although it can be anything. You can think of the primary key as some object that identifies the bean.

> The requirement of a primary key class to identify Entity beans has led to criticism —in particular, by vendors of object-oriented DBMS—that the technology is not particularly object-oriented. In an OODBMS, the object does not need a primary key identifier; it is identified simply by its reference.

Nevertheless, there are some differences between relational entities and Entity beans. Whereas relational modeling requires that the data is normalized, object modeling places no such constraints. Indeed, not even first normal form (where every attribute is scalar) needs to be honored. For example, a `Customer` Entity bean might have a vector attribute called `phoneNumbers`, with a corresponding accessor method `getPhoneNumbers()` that returns a `java.util.List`. In a physical data model, there would need to be a separate table to hold these phone numbers.

Even with a solid logical data model to guide you, selecting Entity beans is not necessarily straightforward. In particular, choosing the granularity of the entities can be problematic. With the customer example given earlier, the `customer_phone` table doesn't really seem significant enough to be an Entity. It's just the way in which vector attributes have to be modeled in relational databases. But what of the invoices? After all, invoices are sent to customers, and any given invoice relates only to the orders placed by a single customer. So perhaps invoices should be considered as just vector attributes of customers, with a `getInvoices()` accessor method? On the other hand, many modelers would argue that the concept of `Invoice` is significant enough in itself—with its own state, behavior, and lifecycle—to warrant being represented as its own Entity bean.

Specifying the interfaces should help you decide which is the correct approach. If the invoice entity really *is* significant, you will find that the customer's interface will be bloated with lots of invoice-related methods. At this point, you can tease the two entity objects apart.

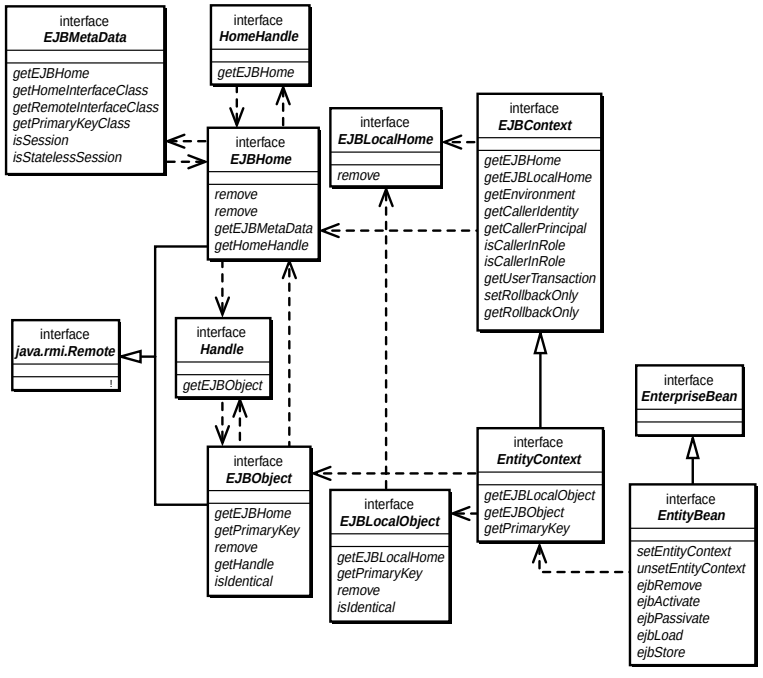
> If you read old text books on EJB design, you will find that the traditional (pre EJB 2.0) advice for Entity beans is that they should be coarse-grained—in other words, that data from several tables correspond to a single entity. This advice arose because of a combination of factors relating to pre EJB 2.0 Entity beans, one in particular being that Entity beans had to be remote (implement the `java.rmi.Remote` interface).
>
> These factors are no longer true, so the advice is out of date. Fine-grained Entity beans are perfectly feasible for an EJB container that supports the EJB 2.0 specification.

The `javax.ejb` Package for Entity Beans

Yesterday, you saw the interfaces and classes in the `javax.ejb` package that related to Session beans. Figure 6.2 shows the interfaces and classes relevant to Entity beans.

FIGURE 6.2

The `javax.ejb` interfaces and classes pertaining to Entity beans.



As you can see, many of the supporting classes are common, which is good news because that means there's less to learn. The principle differences are as follows:

- The Entity bean implements `javax.ejb.EntityBean` rather than `javax.ejb.SessionBean`, so there is a different lifecycle.
- The Entity bean is initialized with an `EntityContext` rather than a `SessionContext`. An `EntityContext` exposes a primary key to the Entity bean, a concept not applicable to Session beans.

Other details of the `javax.ejb` interfaces are the same as for Session beans. Briefly, the home and remote interfaces for the Entity bean are defined by extending `EJBHome` and `EJBObject`, respectively, and the local-home and local interfaces by extending `EJBLocalHome` and `EJBLocalObject`. You will be learning more about local interfaces later today, because they are highly relevant to implementing Entity beans.

The `EJBMetaData` class provides access to the constituent parts of the Entity bean component, and the `Handle` and `HomeHandle` interfaces provide the ability to serialize a reference to a remote bean or home and then to re-instantiate this instance by deserializing the handle. None of these interfaces is discussed further.

Entity Bean Types

Entity beans represent shared persistent data stored in an RDBMS or other persistent data store. If the data store is relational, the responsibility for actually performing the JDBC can be placed either with the bean itself or with the EJB container.

The term for the former is *bean-managed persistence* (BMP), and for the latter it is *container-managed persistence* (CMP).

> The EJB specification is very much oriented around relational data stores. Certainly, container-managed persistence can only be performed through JDBC `javax.sql.DataSource` objects, and JDBC is based around ANSI SQL 92.
>
> If using bean-managed persistence, any API can be used to save the bean's state to a persistent data store, but even then, the methods that the bean is required to provide, such as `findByPrimaryKey()`, still have a relational nature to them.

Container-managed persistence was part of the EJB 1.1 specification (the predecessor to the current EJB 2.0), but attracted much criticism in that release. However, it has been radically overhauled in EJB 2.0, and now works in a fundamentally different way. This is so much so that the deployment descriptor even has the `cmp-version` element to indicate whether the Entity bean has been written under the 1.1 or 2.0 contract. Tomorrow, you will learn more about CMP 2.0. For the rest of today, however, you will be focusing on BMP. That way, you'll have a pleasant surprise when you realize how much of the coding can be automated using CMP.

Remote Versus Local Interfaces

One of the most significant improvements in the EJB 2.0 specification over previous versions is the inclusion of local interfaces as well as remote interfaces.

All beans that you have seen on previous days have provided only a remote interface. That is, both their home and remote interfaces have extended from `javax.ejb.EJBHome` and `javax.ejbEJBObject`, both of which, in turn, extend the `java.rmi.Remote` interface.

This ability to invoke methods on a bean without regard for its location is crucial for Session beans, but for Entity beans it is less useful, even positively harmful. Very often, a client must deal with many Entity beans to transact some piece of work, and if each of those Entity beans is remote, this will incur substantial network traffic. There is also the cost of cloning any serializable objects to enforce the required "pass-by-value" semantics.

Even more frustratingly, the client of the Entity bean may well be a Session bean. Indeed, it is generally considered bad practice to use anything other than a Session bean to interact with Entity beans. More often than not, this Session bean will be co-located (running in the same JVM) as the Entity beans that it is using. The EJB container is obligated to make all Session-to-Entity bean calls via the network and to clone all serializable objects, just because the Entity beans are remote.

By now, you probably have guessed what local interfaces are. They are alternative non-remote interfaces that the Entity bean can specify. Again, the home and proxy idea is used, with the home interface being extended from `javax.ejb.EJBLocalHome` and the proxy for the bean extending from `javax.ejb.EJBLocalObject`. Otherwise though, these are regular Java interfaces, and the normal "pass by reference" semantics for objects passed across these interfaces apply.

"Pass by reference" is a simpler way of saying "object references are passed by value."

An Entity bean can provide a regular home/remote interface, or it can provide a local-home/local interface. Indeed, there is nothing to prevent an Entity bean from offering both interfaces, although any clients using the remote interface would incur the performance costs already noted. Local interfaces are not specific to Entity beans either; Session beans can also provide local interfaces. For Session beans (especially stateless Session beans), there might well be reason to offer both a remote and a local interface. In general, it would be expected for the two interfaces to offer the same sorts of capabilities, although there is nothing in the EJB specification that enforces this.

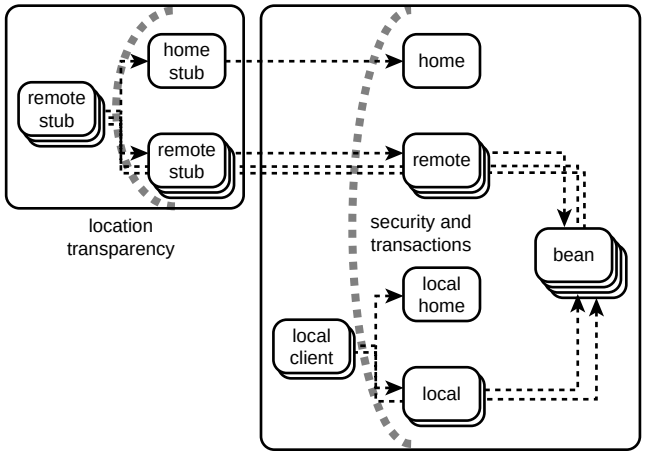
Figure 6.3 shows the two sets of interfaces that a bean can provide.

In both cases, the EJB home/local-home and proxy objects take responsibility for security (Day 15, "Security") and transactions (Day 8, "Transactions and Persistence"), while home/remote interfaces also make the physical location of the bean transparent to the remote client.

Local interfaces are more than just a performance boost for EJBs though, they are the cornerstone on which container-managed persistence and also *container-managed relationships* (CMR) are founded. You will learn about these in detail tomorrow.

FIGURE 6.3

EJBs can have local and remote interfaces.



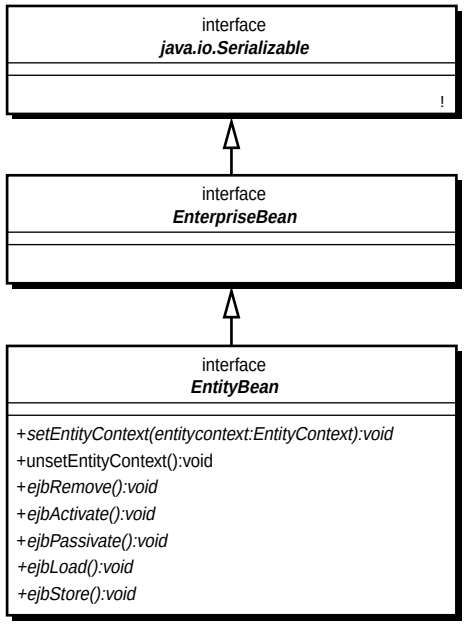
In the case study and examples for today and tomorrow, you will see that the Entity beans define only a local interface.

BMP Entity Bean Lifecycle

The lifecycle of both BMP and CMP Entity beans is dictated by the `EntityBean` interface that the bean must implement. This is shown in Figure 6.4.

FIGURE 6.4

The `javax.ejb.EntityBean` interface defines certain lifecycle methods that must be implemented by Entity beans.



However, although the method names are the same, the obligations of BMP versus CMP Entity beans for each of those methods are different. This section discusses just those lifecycle methods for BMP Entity beans. The `Job` Entity bean from the case study will be predominantly be used for example code.

To start with, the Entity bean must implement the `javax.ejb.EntityBean` interface, as demonstrated with the `JobBean` class:

```
package data;

// imports omitted
import javax.ejb.*;

public class JobBean implements EntityBean
{
    // implementation omitted
}
```

The lifecycle as perceived by the Entity bean and as managed by the container is as shown in Figure 6.5.

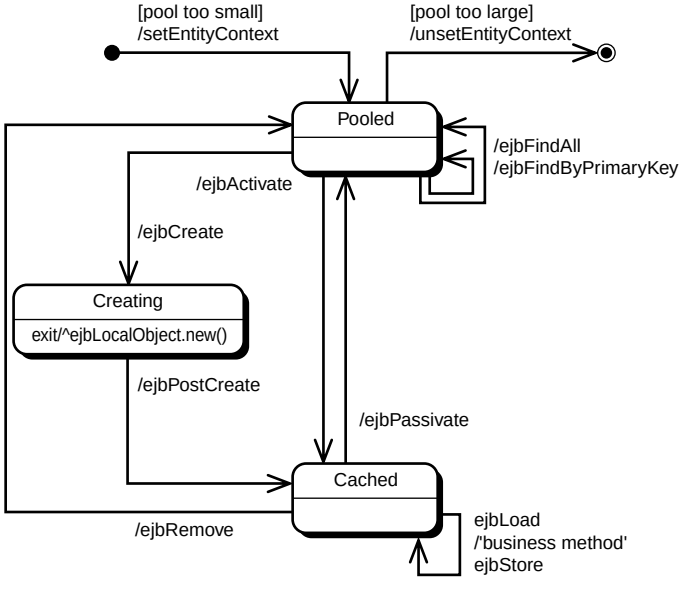


FIGURE 6.5

The `javax.ejb.EntityBean` lifecycle allows Entity beans to be pooled.

The lifecycle is as follows:

- If the EJB container requires an instance of an Entity bean (for example, if the pool is too small), it will instantiate the bean instance and call its `setEntityContext()` method.

- Pooled instances can service finder methods to locate data within the persistent data store that represents existing beans. More on these finder methods shortly.
- A bean can be associated with an `EJBLocalObject` proxy (or `EJBObject` proxy if the remote interface is in use) in one of two ways.

  First, it can be activated by the container via `ejbActivate()`. The proxy for the bean exists but has no associated bean. This could occur if the bean had previously been passivated and a business method has now been invoked on the proxy. It could also occur if the bean's proxy was just returned as the result of a finder method.

  Alternatively, the client may be requesting to create an Entity bean via `ejbCreate()` and then `ejbPostCreate()`. This usually means that the corresponding data has been inserted into the persistent data store.
- When the bean has been associated with its proxy, business methods can be invoked on it. Before the business method is delegated by the proxy to the bean, the `ejbLoad()` lifecycle method will be called, indicating that the bean should reload its state from the persistent data store. Immediately after the business method has completed, the `ejbStore()` method is called, indicating that the bean should update the persistent data store with any change in its state.
- Beans can return to the pooled state in one of two ways.

  First, they can be passivated via `ejbPassivate()`. There is usually little to be done in the lifecycle, because the bean's state will already have been saved to the persistent data store during the earlier `ejbStore()` method. So passivation simply means that the link from the `EJBLocalObject` proxy to the bean has been severed.

  Alternatively, the client may be requesting to remove the create bean via `ejbRemove()`. This usually means that the corresponding data in the persistent data store has been deleted.
- Finally, if the EJB container wants, it can reduce the size of its pool by first calling `unsetEntityContext()`.

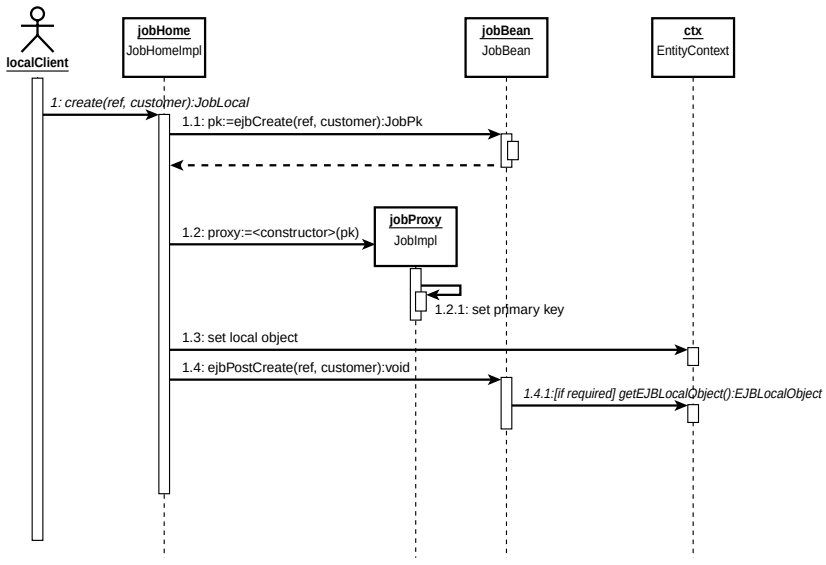
> Most commercial EJB containers provide mechanisms to suppress unnecessary `ejbLoad()` and `ejbStore()` calls. None of these mechanisms are in the EJB specification, however.

Unlike Session beans, there is no binding of the Entity beans to a specific client; the bean can be shared by all clients.

As Figure 6.5 indicated, there are two methods called during the creation of a bean. The `ejbCreate()` method is called prior to the `EJBLocalObject` proxy being made available, the `ejbPostCreate()` method is called after the proxy is available. This is shown in the sequence diagram in Figure 6.6.

FIGURE 6.6

Both the `ejbCreate()` and `ejbPostCreate()` lifecycle methods are called when an Entity bean is created.



Under BMP, the bean has several tasks to perform when its `ejbCreate()` method is called. It should:

- Calculate the value of its primary key (if not passed in as an argument).
- Persist itself to a data store. For a RDBMS, this will most likely be in the form of an SQL `INSERT` statement or statements.
- Save the supplied arguments and its primary key to fields.
- Return the primary key.

As Figure 6.6 shows, the returned primary key is passed to the bean's proxy, and the proxy continues to hold that primary key, even if the bean is subsequently passivated. The proxy for the bean is also associated with the context object of the bean.

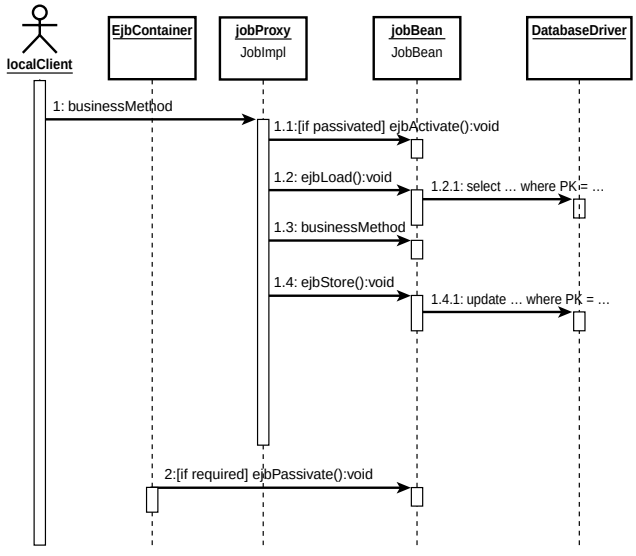
You can see that the `EJBLocalObject` proxy holds onto the primary key for the bean. This allows the bean to be transparently re-loaded if it is passivated. However, because the EJB container is using primary keys for lookups, it also means the EJB does not allow primary keys to be modified by the application; they must be immutable.

The `ejbRemove()` method is the opposite of the `ejbCreate()` method; it removes a bean's data from the persistent data store. The implementation of `ejbCreate()` and `ejbRemove()` is given in the "Implementing `javax.ejb.EntityBean`" section later today.

That takes care of creating and removing beans, but what about when a bean is queried or updated? The most significant of the Entity bean lifecycle methods are the `ejbLoad()` and `ejbStore()`methods. Together, these methods ensure that the Entity bean is kept in sync with the persistent data store. The `ejbLoad()` method is called immediately prior to any business method (so that a query access the most up-to-date data). The `ejbStore()` is called after the business method completes (so that if the method updated the bean's state, this is reflected in the persistent data store). Figure 6.7 shows this as a UML sequence diagram.

FIGURE 6.7

The `ejbLoad()` and `ejbStore()` methods keep the bean in sync with the persistent data store.



Again, the actual implementation for these methods is given in the "Implementing `javax.ejb.EntityBean`" section later today.

As you will recall, Session beans have `ejbActivate()` and `ejbPassivate()` methods, and so do Entity beans. If the EJB container wants to reduce the number of bean instances, it can passivate the bean. This is only ever done after an `ejbStore()`, so the data represented by the bean is not lost. Also, the proxy for the bean continues to hold the bean's primary key, meaning that if the client interacts with the bean (through the proxy) in the future, the appropriate data can be loaded from the persistent data store. Generally, then, there is little or nothing to be done when an Entity bean is passivated or activated.

These lifecycle methods allow new beans (and data in the persistent data store) to be created or removed and updating existing beans, but what about actually finding beans that already exist? In other words, in JDBC terms, you have seen the lifecycle methods that correspond to SQL `INSERT`, `DELETE`, and `UPDATE` statements, but what of an SQL `SELECT` statement? Well, this is accomplished by the finder methods. The EJB specification requires at least one finder method, whose name must be `ejbFindByPrimaryKey()`, and allows other finder methods, whose names must begin `ejbFind`. These methods have corresponding methods in the local-home interface, so you'll be learning about them shortly as part of specifying and implementing the bean.

One obvious question arises, "When the client invokes the finder method on the home interface, which bean actually performs the `ejbFindXxx()` method?" The answer is perhaps a little unexpected; any unused (that is, pooled) bean will be used by the EJB container.

Learning all these lifecycle methods for both Entity and Session beans can be somewhat overwhelming at first, made all the more complicated because some method names appear for both bean types but imply different responsibilities. To clarify matters, Table 6.1 compares the two sets of lifecycle methods and identifies those responsibilities.

TABLE 6.1 Responsibilities of Session and Entity Beans Sit in Different Lifecycle Methods

Lifecycle Method	*Session Bean*	*Entity Bean*
`setXxxContext()`	Set context	Set context
`unsetXxxContext()`	N/A	Unset context
`ejbFindByPrimaryKey()`	N/A	Acquire reference to proxy
`ejbCreate()`	Acquire reference to proxy	a) Insert data to persistent data store b) Acquire reference to proxy
`ejbPostCreate()`	N/A	Access proxy if necessary
`ejbActivate()`	a) Loaded from (temporary) data store b) Obtain environmental resources	Obtain environmental resources
`ejbPassivate()`	a) Saved to (temporary) data store; b) Release environmental resources	Release environmental resources
`ejbLoad()`	N/A	Load from (persistent) data store
`ejbStore()`	N/A	Save to (persistent) data store
`ejbRemove()`	Release reference to proxy	a) Delete data from persistent data store b) Release reference to proxy

Specifying a BMP Entity Bean

Following the pattern of Session beans, specifying an Entity bean involves defining the local-home and the local interface:

- The local-home interface extends `javax.ejb.EJBLocalHome`.
- The local interface extends `javax.ejb.EJBLocalObject`.

A discussion on each of these interfaces follows.

Local-Home Interface

Listing 6.1 shows the complete `JobLocalHome` interface as an example.

LISTING 6.1 `JobLocalHome` Interface

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import java.util.*;
 5: import javax.ejb.*;
 6:
 7: public interface JobLocalHome extends EJBLocalHome
 8: {
 9:      JobLocal create (String ref, String customer) throws CreateException;
10:      JobLocal findByPrimaryKey(JobPK key) throws FinderException;
11:      Collection findByCustomer(String customer) throws FinderException;
12:      Collection findByLocation(String location) throws FinderException;
13:      void deleteByCustomer(String customer);
14: }
```

Each of these methods has a corresponding method in the bean class itself. Taking the `JobBean` code as an example:

- The `create(String ref, String customer)` method in `JobBean` corresponds to `ejbCreate(String ref, String customer)` in the `JobLocalHome` interface.
- The `ejbFindByPrimaryKey(String name)` method in JobBean corresponds to `findByPrimaryKey(String name)` in the `JobLocalHome` interface.
- The `ejbFindByCustomer(String customer)` method in `JobBean` corresponds to `findbyCustomer(String customer)` in the `JobLocalHome` interface.
- The `ejbHomeDeleteByCustomer(String customer)` in `JobBean` corresponds to `deleteByCustomer(String customer)` in the `JobLocalHome` interface.

Note that for home methods discussed shortly, the convention is to append `ejbHome`, not just `ejb`, to the bean's method name.

This seems straight-forward enough, but note that the return types for the bean's `ejbCreate()` and `ejbFindXxx()` methods are different from the return types of the methods in the local-home interface. Specifically, while the bean returns (to the EJB container) either primary key objects or `Collections` of primary key objects, the local-home interface methods return (to the client) either local proxies (that is, instances of objects that implement the `JobLocal` interface, for the example) or `Collections` of such.

Create and Remove Methods

The list of exceptions thrown by the local-home methods and the bean's corresponding methods should match in each case. For the `createXXX()` method, the list should be the union of the exceptions thrown by both `ejbCreateXXX()` and `ejbPostCreateXXX()`. If a home and a remote interface are being provided for the Entity bean, the `java.rmi.RemoteException` must be declared for the methods of the home interface.

As well as the `create()` method, the local-home interface inherits a `remove(Object o)` method from `javax.ejb.EJBLocalHome`. This corresponds to the `ejbRemove()` lifecycle method of the bean itself.

Finder Methods

Finder methods in the bean return either a single primary key (if a single bean matches the underlying query) or a `java.util.Collection` of primary keys (if there is more than one matching bean). The `ejbFindByPrimaryKey()` method is always required to be one of the bean's methods, although it is not part of the `EntityBean` interface. This is because the argument type and return type will depend upon the bean.

It is also possible for finder methods to return `java.util.Enumerations`. This dates from EJB 1.0 before the Java Collections API was introduced in J2SE 1.2 and should not be used.

Obviously, to specify the `findByPrimaryKey()` method, the primary key of the Entity bean must have been identified. As was noted earlier today, if persisting to an RDBMS, identifying the primary key is probably quite easy, because the primary key will correspond to the columns of the primary key of the underlying RDBMS table. A custom-developed primary key class is needed when two or more fields identify the bean; otherwise, the type of the single field of the bean that represents the key is used.

If a single field of the bean is used as the primary key, that field must not be a primitive type (such as an `int` or `long`). Primary key fields must be actual classes, such as a `java.lang.String`. Furthermore, the EJB specification does not allow primary keys to change once assigned, so it is best if the class chosen is immutable.

Custom Primary Key Classes

As noted earlier, the primary key can be either a field of the bean (in which case, the primary key class is just the class of that field) or can be a custom-developed class. The latter is required if more than one field is needed to identify the bean (and can be used even for single field keys).

For the `JobBean`, the primary key is a combination of the customer and the job reference (the `customer` and `ref` fields, respectively). Because the primary key is composite, a custom primary key class is needed; this is the `JobPK` class.

Custom primary key classes are required to follow a number of rules. Specifically

- The class must implement `java.io.Serializable` or `java.io.Externalizable`.
- The values of the class must all be primitives or be references to objects that, in turn, are serializable.
- The `equals()` method and the `hashCode()` methods must be implemented.
- There must be a no-arg constructor (there can also be other constructors that take arguments, but they would only be provided for convenience).

In other words, the class must be what is sometimes referred to as a *value type*.

At least conceptually, value types are immutable (there should be no setter methods; they cannot be changed). The requirement for a no-arg constructor does prevent this from actually being the case.

Listing 6.2 shows the `JobPK` primary key class.

LISTING 6.2 `JobPK` Class Identifies a `Job`

```
1: package data;
2:
3: import java.io.*;
4: import javax.ejb.*;
5:
```

LISTING 6.2 Continued

```
 6: public class JobPK implements Serializable {
 7:     public String ref;
 8:     public String customer;
 9:
10:     public JobPK() {
11:     }
12:     public JobPK(String ref, String customer) {
13:         this.ref = ref;
14:         this.customer = customer;
15:     }
16:
17:     public String getRef() {
18:         return ref;
19:     }
20:     public String getCustomer() {
21:         return customer;
22:     }
23:
24:     public boolean equals(Object obj) {
25:         if (obj instanceof JobPK) {
26:             JobPK pk = (JobPK)obj;
27:             return (pk.ref.equals(ref) && pk.customer.equals(customer));
28:         }
29:         return false;
30:     }
31:     public int hashCode() {
32:         return (ref.hashCode() ^ customer.hashCode());
33:     }
34:     public String toString() {
35:         return "JobPK: ref=\"" + ref + "\", customer=\"" +
➥ customer + "\"";
36:     }
37: }
```

Note that the `ref` and `customer` fields have `public` visibility. This is a requirement of the EJB specification. Each field must correspond—in name and type—to one of the fields of the bean itself. This might seem like a strange requirement, but is needed by the EJB container to manage CMP beans.

To implement the `equals()` method, test that all fields of the object have the same value as the fields in the provided object. For primitive values, the regular `==` operator should be used, but for object references, the `equals()` method must be called.

To implement the `hashCode()` method, generate an `int` value that is based entirely and deterministically on the value of the fields, such that

if `A.equals(B)` then `A.hashCode() == B.hashCode()`.

There are a couple of ways to accomplish this. A quick way to do this is to convert all the values of the primary key class' fields to `String`s, concatenate them to a single `String`, and then invoke the `hashCode()` on this resultant string. Alternatively, the `hashCode()` values for all of the fields could be `or`'d together using the `^` operator. At runtime, this will execute more quickly than the concatenation approach, but it does mean that the distribution of hashcodes may be less good for primary keys with many fields. This is the approach used in Listing 6.2.

Creating these primary key classes can be somewhat tedious. But remember that if there is a single (non-primitive) field in the bean that identifies that bean, this can be used instead.

Failing that, a single primary key class can be used for multiple beans. For example, you could create a `IntPK` class that just encapsulates an `int` primitive value.

Home Methods

In addition to finder, create, and remove methods, it is also possible to define home methods within the local-home interface. These are arbitrary methods that are expected to perform some business-type functionality related to the set of beans. In other words, they are an EJB equivalent of Java class methods (defined with the `static` keyword).

Some common uses for home methods include defining a batch operation to be performed on all bean instances (such as decreasing the price of all catalogue items for a sale), or various utility methods, such as formatting a bean's state for a `toString()` method.

One question that sometimes arises is whether all database updates should be performed through Entity bean methods. One example given for a home method of a bean would be to decrease the price of all catalogue items for a sale. Iterating over perhaps 10,000 catalogue items and invoking `setPrice( getPrice() * 0.9 )` is clearly going to cause massive amounts of SQL hitting the back-end RDBMS (or equivalent persistent data store).

In J2SE programs, a simple update, such as

```
UPDATE catalogue
set price = price * 0.9
```

is clearly the way to go. The `ejbLoad()` lifecycle method will ensure that any catalog item Entity bean will re-sync its state with the RDBMS.

Local Interface

Just as for Session beans with their remote interfaces, the local interface defines the capabilities of the Entity bean. Because first and foremost an Entity bean represents data, it is entirely to be expected that many of the methods exposed through the local interface will be simple getter and setter methods. Listing 6.3 shows the local interface for the `Job` bean.

LISTING 6.3 `JobLocal` Interface

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import javax.ejb.*;
 5:
 6: public interface JobLocal extends EJBLocalObject
 7: {
 8:     String getRef();
 9:     String getCustomer();
10:     CustomerLocal getCustomerObj(); // derived
11:
12:     void setDescription(String description);
13:     String getDescription();
14:
15:     void setLocation(LocationLocal location);
16:     LocationLocal getLocation();
17:
18:     Collection getSkills();
19:     void setSkills(Collection skills);
20: }
```

Note that the `setLocation()` method accepts a `LocationLocal` reference rather than, say, a `String` containing the name of a location. In other words, the `Job` bean is defining its relationships to other beans, in this case the `Location` bean directly, effectively enforcing referential integrity. The client of the `Job` Entity bean is thus required to supply a valid location or none at all.

This is not to say that Entity beans cannot provide further processing. An example often quoted might be for a `SavingsAccountBean`. This might provide `withdraw()` and `deposit()` methods. The `withdraw()` method might well ensure that the balance can never go below zero. The bean might also provide an `applyInterest()` method, but it almost certainly would not provide a `setBalance()` method (if only!).

Each of these methods has a corresponding method in the bean itself, and the exceptions list matches exactly. The implementation is shown in the “Implementing the Local-Interface Methods” section later today.

Actually, such an `SavingsAccountBean` might well provide a `setBalance()` method, but would restrict access to administrators. You will learn more about security on Day 15, "Security."

Implementing a BMP Entity Bean

Implementing an Entity bean involves providing an implementation for the methods of the `javax.ejb.EntityBean`, corresponding methods for each method in the local-home interface, and a method for each method in the local interface.

Implementing javax.ejb.EntityBean

The `setEntityContext()` method is a good place to perform JNDI lookups, for example to acquire a JDBC DataSource reference. Listing 6.4 shows how this is done for the `JobBean` code.

LISTING 6.4 `JobBean.setEntityContext()` Method

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import javax.naming.*;
 5: import javax.sql.*;
 6: // imports omitted
 7:
 8: public class JobBean implements EntityBean
 9: {
10:     public void setEntityContext(EntityContext ctx) {
11:         this.ctx = ctx;
12:         InitialContext ic = null;
13:         try {
14:             ic = new InitialContext();
15:             dataSource = (DataSource)
               ➥ic.lookup("java:comp/env/jdbc/Agency");
16:             skillHome = (SkillLocalHome)
               ➥ic.lookup("java:comp/env/ejb/SkillLocal");
17:             locationHome = (LocationLocalHome)
               ➥ic.lookup("java:comp/env/ejb/LocationLocal");
18:             customerHome = (CustomerLocalHome)
               ➥ic.lookup("java:comp/env/ejb/CustomerLocal");
19:         }
20:         catch (NamingException ex) {
21:             error("Error looking up depended EJB or resource",ex);
22:             return;
23:         }
24:   }
25:
```

LISTING 6.4 Continued

```
26:     private Context ctx;
27:     private DataSource dataSource
28:
29:     // code omitted
30: }
```

The `unsetEntityContext()` method (not shown) usually just sets these fields to `null`.

The `ejbLoad()` and `ejbStore()` methods are responsible for synchronizing the bean's state with the persistent data store. Listing 6.5 shows these methods for `JobBean`.

LISTING 6.5 `JobBean`'s `ejbLoad()` and `ejbStore()` Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import java.sql.*;
 5: // imports omitted
 6:
 7: public class JobBean implements EntityBean
 8: {
 9:     public void ejbLoad(){
10:         JobPK key = (JobPK)ctx.getPrimaryKey();
11:         Connection con = null;
12:         PreparedStatement stmt = null;
13:         ResultSet rs = null;
14:         try {
15:             con = dataSource.getConnection();
16:             stmt = con.prepareStatement(
                ➥"SELECT description,location
                ➥ FROM Job
                ➥ WHERE ref = ? AND customer = ?");
17:             stmt.setString(1, key.getRef());
18:             stmt.setString(2, key.getCustomer());
19:             rs = stmt.executeQuery();
20:             if (!rs.next()) {
21:                 error("No data found in ejbLoad for " + key, null);
22:             }
23:             this.ref = key.getRef();
24:             this.customer = key.getCustomer();
25:             this.customerObj =
                ➥customerHome.findByPrimaryKey(this.customer); // derived
26:             this.description = rs.getString(1);
27:             String locationName = rs.getString(2);
28:             this.location = (locationName != null) ?
                ➥locationHome.findByPrimaryKey(locationName) : null;
29:             // load skills
```

LISTING 6.5 Continued

```
30:             stmt = con.prepareStatement(
                ➥"SELECT job, customer, skill
                ➥ FROM JobSkill
                ➥ WHERE job = ? AND customer = ?
                ➥ ORDER BY skill");
31:             stmt.setString(1, ref);
32:             stmt.setString(2, customerObj.getLogin());
33:             rs = stmt.executeQuery();
34:             List skillNameList = new ArrayList();
35:             while (rs.next()) {
36:                 skillNameList.add(rs.getString(3));
37:             }
38:             this.skills = skillHome.lookup(skillNameList);
39:         }
40:         catch (SQLException e) {
41:             error("Error in ejbLoad for " + key, e);
42:         }
43:         catch (FinderException e) {
44:             error("Error in ejbLoad (invalid customer or location) for "
                      ➥+ key, e);
45:         }
46:         finally {
47:             closeConnection(con, stmt, rs);
48:         }
49:     }
50:
51:     public void ejbStore(){
52:         Connection con = null;
53:         PreparedStatement stmt = null;
54:         try {
55:             con = dataSource.getConnection();
56:             stmt = con.prepareStatement(
                ➥"UPDATE Job
                ➥ SET description = ?, location = ?
                ➥ WHERE ref = ? AND customer = ?");
57:             stmt.setString(1, description);
58:             if (location != null) {
59:                 stmt.setString(2, location.getName());
60:             } else {
61:                 stmt.setNull(2, java.sql.Types.VARCHAR);
62:             }
63:             stmt.setString(3, ref);
64:             stmt.setString(4, customerObj.getLogin());
65:             stmt.executeUpdate();
66:             // delete all skills
67:             stmt = con.prepareStatement(
                ➥"DELETE FROM JobSkill
                ➥ WHERE job = ? and customer = ?");
```

LISTING 6.5 Continued

```
68:             stmt.setString(1, ref);
69:             stmt.setString(2, customerObj.getLogin());
70:             stmt.executeUpdate();
71:             // add back in all skills
72:             for (Iterator iter = getSkills().iterator(); iter.hasNext();){
73:                 SkillLocal skill = (SkillLocal)iter.next();
74:                 stmt = con.prepareStatement(
                    ➥"INSERT INTO JobSkill (job,customer,skill)
                    ➥ VALUES (?,?,?)");
75:                 stmt.setString(1, ref);
76:                 stmt.setString(2, customerObj.getLogin());
77:                 stmt.setString(3, skill.getName());
78:                 stmt.executeUpdate();
79:             }
80:         }
81:         catch (SQLException e) {
82:             error("Error in ejbStore for " + ref + "," + customer, e);
83:         }
84:         finally {
85:             closeConnection(con, stmt, null);
86:         }
87:     }
88:     // code omitted
89: }
```

In the `ejbLoad()` method, the `JobBean` must load its state from both the `Job` and `JobSkill` tables, using the data in the `JobSkill` table to populate the `skills` field. In the `ejbStore()` method, the equivalent updates to the `Job` and `JobSkill` tables occur.

Of course, there is the chance that when the bean comes to save itself, the data could have been removed. This would happen if some user manually deleted the data; there is nothing in the EJB specification to require that an Entity bean "locks" the underlying data. In such a case, the bean should throw a `javax.ejb.NoSuchEntityException`; in turn, this will be returned to the client as some type of `java.rmi.RemoteException`. This was mentioned briefly yesterday, so look back to refresh your memory if needed. And remember, you will be learning more about exception handling and transactions on Day 8.

To keep the case study as small and understandable as possible, the error handling in `JobBean` is slightly simplified. In Listing 6.5, the code will throw an `EJBException` (rather than `NoSuchEntityException`) from `ejbLoad()` if the data has been removed. In `ejbStore()`, it doesn't actually check to see if any rows were updated, so no exception would be thrown.

More complex beans can perform other processing within the `ejbLoad()` and `ejbStore()` methods. For example, the data might be stored in some denormalized form in a relational database, perhaps for performance reasons. The `ejbStore()` method would store the data in this de-normalized form, while the `ejbLoad()` methods would effectively be able to re-normalize the data on-the-fly. The client need not be aware of these persistence issues.

Another idea: these methods could be used to handle text more effectively. The EJB specification suggests compressing and decompressing text, but they could also perhaps do searches for keywords within the text, and then redundantly store these keywords separately, or the data might be converted into XML format.

As noted earlier today, there is usually very little or nothing to be done when an Entity bean is passivated or activated. Listing 6.6 shows this.

LISTING 6.6 `JobBean`'s `ejbActivate()` and `ejbPassivate()` Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: // imports omitted
 5:
 6: public class JobBean implements EntityBean
 7: {
 8:     public void ejbPassivate(){
 9:         ref = null;
10:         customer = null;
11:         customerObj = null;
12:         description = null;
13:         location = null;
14:     }
15:
16:     public void ejbActivate(){
17:     }
18:
19:     // code omitted
20: }
```


Implementing the Local-Home Interface Methods

The implementation of `ejbCreate()` and `ejbPostCreate()` for the `JobBean` is shown in Listing 6.7.

LISTING 6.7 `JobBean`'s `ejbCreate()` and `ejbPostCreate()` Methods

```
 1: package data;
 2:
```

LISTING 6.7 Continued

```
 3: import javax.ejb.*;
 4: import javax.sql.*;
 5: // imports omitted
 6:
 7: public class JobBean implements EntityBean
 8: {
 9:     private String ref;
10:     private String customer;
11:     private String description;
12:     private LocationLocal location;
13:     private CustomerLocal customerObj; // derived
14:     private List skills;    // vector field; list of SkillLocal ref's.
15:
16:     public String ejbCreate (String ref, String customer)
                  ➥throws CreateException {
17:         // validate customer login is valid.
18:         try {
19:             customerObj = customerHome.findByPrimaryKey(customer);
20:         } catch (FinderException ex) {
21:             error("Invalid customer.", ex);
22:         }
23:         JobPK key = new JobPK(ref, customer);
24:         try {
25:             ejbFindByPrimaryKey(key);
26:             throw new CreateException("Duplicate job name: " + key);
27:         }
28:         catch (FinderException ex) { }
29:         Connection con = null;
30:         PreparedStatement stmt = null;
31:         try {
32:             con = dataSource.getConnection();
33:             stmt = con.prepareStatement(
                ➥"INSERT INTO Job (ref,customer)
                ➥ VALUES (?,?)");
34:             stmt.setString(1, ref);
35:             stmt.setString(2, customerObj.getLogin());
36:             stmt.executeUpdate();
37:         }
38:         catch (SQLException e) {
39:             error("Error creating job " + key, e);
40:         }
41:         finally {
42:             closeConnection(con, stmt, null);
43:         }
44:         this.ref = ref;
45:         this.customer = customer;
46:         this.description = description;
47:         this.location = null;
48:         this.skills = new ArrayList();
```

LISTING 6.7 Continued

```
49:         return key;
50:     }
51:
52:     public void ejbPostCreate (String name, String description) {}
53: }
```

This particular implementation validates that the customer exists (jobs are identified by customer and by a unique reference), and it also makes sure that the full primary key does not already exist in the database. If it does, the BMP bean throws a `CreateException`. If it doesn't (represented by the `ejbFindByPrimaryKey()` call throwing a `FinderException`), the method continues.

An alternative implementation would have been to place a unique index on the `Job` table within the RDBMS and then to catch the `SQLException` that might be thrown if a duplicate is attempted to be inserted.

There is a race condition here. It's possible that another user could insert a record between the check for duplicates and the actual SQL `INSERT`. The `ejbCreate()` method is called within a transaction; changing the RDBMS isolation level (in a manner specified by the EJB container) would eliminate this risk, although deadlocks could then occur.

Note that the `skills` field is set to an empty `ArrayList`. This holds a list of `SkillLocal` references, this being the local interface to the `Skill` bean. Of course, for a newly created `Job` bean, this list is empty. The decision for the `skills` field to hold references to `SkillLocal` objects rather than, say, just `String`s holding the skill names, was taken advisedly. If the skill name is used (that is, the primary key of a skill), finding information about the skill would require extra steps. Perhaps more compellingly, this is also the approach taken for CMP beans and container-managed relationships, discussed in detail tomorrow.

Also noteworthy is the `customerObj` field. The `Job`, when created, is passed just a `String` containing the customer's name. In other words, this is a primary key to a customer. The `customerObj` field contains a reference to the parent customer bean itself by way of its `CustomerLocal` reference.

Both the `skills` and the `customerObj` fields illustrate (for want of a better phrase) bean-managed relationships. For the `skills` field, this is a many-to-many relationship, from `Job` to `Skill`. For the `customerObj` field, this is a many-to-one relationship from `Job` to `Customer`.

As for the stateful Session beans that you learned about yesterday, the `ejbCreate()` and `ejbPostCreate()` methods both correspond to a single method called `create()` in the bean's local-home interface. The list of arguments must correspond. Again, as for Session beans, it is possible for there to be more than one create method with different sets of arguments, or indeed the `create`*`XXX`*`()` method naming convention can be used instead of overloading the method name of `create()`.

The `ejbRemove()` method is the opposite of the `ejbCreate()` method; it removes a bean's data from the persistent data store. Its implementation for `JobBean` is shown in Listing 6.8.

LISTING 6.8 `JobBean`'s `ejbRemove()` Method

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import javax.naming.*;
 5: // imports omitted
 6:
 7: public class JobBean implements EntityBean
 8: {
 9:     public void ejbRemove(){
10:         JobPK key = (JobPK)ctx.getPrimaryKey();
11:         Connection con = null;
12:         PreparedStatement stmt1 = null;
13:         PreparedStatement stmt2 = null;
14:         try {
15:             con = dataSource.getConnection();
16:             stmt1 = con.prepareStatement(
                ➥"DELETE FROM JobSkill
                ➥ WHERE job = ? and customer = ?");
17:             stmt1.setString(1, ref);
18:             stmt1.setString(2, customerObj.getLogin());
19:             stmt2 = con.prepareStatement(
                ➥"DELETE FROM Job
                ➥ WHERE ref = ? and customer = ?");
20:             stmt2.setString(1, ref);
21:             stmt2.setString(2, customerObj.getLogin());
22:             stmt1.executeUpdate();
23:             stmt2.executeUpdate();
24:         }
25:         catch (SQLException e) {
26:             error("Error removing job " + key, e);
27:         }
28:         finally {
29:             closeConnection(con, stmt1, null);
30:             closeConnection(con, stmt2, null);
31:         }
32:         ref = null;
```

LISTING 6.8 Continued

```
33:         customer = null;
34:         customerObj = null;
35:         description = null;
36:         location = null;
37:     }
38:     // code omitted
39: }
```

Each of the finder methods of the local-home interface must have a corresponding method in the bean. By way of example, Listing 6.9 shows two of the (three) finder methods for the `JobBean`.

LISTING 6.9 `JobBean`'s Finder Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import java.sql.*;
 5: import java.util.*;
 6: // imports omitted
 7:
 8: public class JobBean implements EntityBean
 9: {
10:     public JobPK ejbFindByPrimaryKey(JobPK key) throws FinderException {
11:         Connection con = null;
12:         PreparedStatement stmt = null;
13:         ResultSet rs = null;
14:         try {
15:             con = dataSource.getConnection();
16:             stmt = con.prepareStatement(
                ➥"SELECT ref
                ➥ FROM Job
                ➥ WHERE ref = ? AND customer = ?");
17:             stmt.setString(1, key.getRef());
18:             stmt.setString(2, key.getCustomer());
19:             rs = stmt.executeQuery();
20:             if (!rs.next()) {
21:                 throw new FinderException("Unknown job: " + key);
22:             }
23:             return key;
24:         }
25:         catch (SQLException e) {
26:             error("Error in findByPrimaryKey for " + key, e);
27:         }
28:         finally {
29:             closeConnection(con, stmt, rs);
30:         }
31:         return null;
```

LISTING 6.9 Continued

```
32:     }
33:
34:     public Collection ejbFindByCustomer(String customer)
                    ➥throws FinderException {
35:         Connection con = null;
36:         PreparedStatement stmt = null;
37:         ResultSet rs = null;
38:         try {
39:             con = dataSource.getConnection();
40:             stmt = con.prepareStatement(
                ➥"SELECT ref, customer
                ➥ FROM Job
                ➥ WHERE customer = ?
                ➥ ORDER BY ref");
41:             stmt.setString(1, customer);
42:             rs = stmt.executeQuery();
43:             Collection col = new ArrayList();
44:             while (rs.next()) {
45:                 String nextRef = rs.getString(1);
46:                 String nextCustomer = rs.getString(2);
47:                 // validate customer exists
48:                 CustomerLocal nextCustomerObj =
                    ➥customerHome.findByPrimaryKey(nextCustomer);
49:                 col.add(new JobPK(nextRef, nextCustomerObj.getLogin()));
50:             }
51:             return col;
52:         }
53:         catch (SQLException e) {
54:             error("Error in findByCustomer: " + customer, e);
55:         }
56:         catch (FinderException e) {
57:             error("Error in findByCustomer, invalid customer: " +
                        ➥customer, e);
58:         }
59:         finally {
60:             closeConnection(con, stmt, rs);
61:         }
62:         return null;
63:     }
64:
65:    // code omitted
66: }
```

The implementation of the `ejbFindByPrimaryKey()` method might seem somewhat unusual; it receives a primary key, and then returns it. Of course, what it has done as well is to have validated that an entity exists for the given primary key; if there were none, a `javax.ejb.ObjectNotFoundException` would be thrown. The implementation of `ejbFindByCustomer()` is straightforward enough.

The `Job` bean defines a home method, namely `deleteByCustomer()`, and the corresponding method in the `JobBean` class is `ejbHomeDeleteByCustomer()`, as shown in Listing 6.10.

LISTING 6.10 `JobBean.ejbHomeDeleteByCustomer()` Home Method

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import java.sql.*;
 5: import java.util.*;
 6: // imports omitted
 7:
 8: public class JobBean implements EntityBean
 9: {
10:     public void ejbHomeDeleteByCustomer(String customer) {
11:         Connection con = null;
12:         PreparedStatement stmt2 = null;
13:         PreparedStatement stmt1 = null;
14:         try {
15:             con = dataSource.getConnection();
16:             stmt1 = con.prepareStatement(
                ➥"DELETE FROM JobSkill
                ➥ WHERE customer = ?");
17:             stmt2 = con.prepareStatement(
                ➥"DELETE FROM Job
                ➥ WHERE customer = ?");
18:             stmt1.setString(1, customer);
19:             stmt2.setString(1, customer);
20:             stmt1.executeUpdate();
21:             stmt2.executeUpdate();
22:         }
23:         catch (SQLException e) {
24:             error("Error removing all jobs for " + customer, e);
25:         }
26:         finally {
27:             closeConnection(con, stmt1, null);
28:             closeConnection(con, stmt2, null);
29:         }
30:     }
31:    // code omitted
32: }
```

Implementing the Local Interface Methods

Each of the methods in the local interface has a corresponding method in the bean itself. The corresponding methods for `JobBean` are shown in Listing 6.11.

LISTING 6.11 Business Methods of `JobBean` Correspond to the Methods of the Local Interface

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import javax.ejb.*;
 5: // imports omitted
 6:
 7: public class JobBean implements EntityBean
 8: {
 9:     public String getRef() {
10:         return ref;
11:     }
12:     public String getCustomer() {
13:         return customer;
14:     }
15:     public CustomerLocal getCustomerObj() {
16:         return customerObj;
17:     }
18:     public String getDescription() {
19:         return description;
20:     }
21:     public void setDescription(String description) {
22:         this.description = description;
23:     }
24:     public LocationLocal getLocation() {
25:         return location;
26:     }
27:     public void setLocation(LocationLocal location) {
28:         this.location = location;
29:     }
30:     /** returns (copy of) skills */
31:     public Collection getSkills() {
32:         return new ArrayList(skills);
33:     }
34:     public void setSkills(Collection skills) {
35:         // just validate that the collection holds references to
               ➥SkillLocal's
36:         for (Iterator iter = getSkills().iterator(); iter.hasNext(); ) {
37:             SkillLocal skill = (SkillLocal)iter.next();
38:         }
39:         // replace the list of skills with that defined.
40:         this.skills = new ArrayList(skills);
41:     }
42:    // code omitted
43: }
```

The `getSkills()` and `setSkills()` methods bear closer inspection. The `getSkills()` method returns a copy of the local `skills` field because, otherwise, the client could

change the contents of the `skills` field without the `JobBean` knowing. This isn't an issue that would have arisen if the interface to `JobBean` was remote, because a copy would automatically have been created. Turning to the `setSkills()` method, this checks to make sure that the new collection of skills supplied is a list of `SkillLocal` references. This is analogous to the `setLocation()` method that was discussed; the `Job` Entity bean is enforcing referential integrity with the `Skill` Entity bean.

Generating IDs

Sometimes an Entity bean already has a field (or fields) that represent the primary key, but at other times, the set of fields required may just be too large. Alternatively, the obvious primary key may not be stable in the sense that its value could change over the lifetime of an entity—something prohibited by the EJB specification. For example, choosing a (*`lastname`*, *`firstname`*) as a means of identifying an employee may fail if a female employee gets married and chooses to adopt her husband's surname.

In these cases, it is common to introduce an artificial key, sometimes known as a *surrogate key*. You will be familiar with these if you have ever been allocated a customer number when shopping online. Your social security number, library card number, driver's license, and so on may well be just a pretty meaningless jumble of numbers and letters, but they are guaranteed to be unique and stable. These are all surrogate keys.

With BMP Entity beans, the responsibility for generating these ID values is yours, the bean provider. Whether numbers and letters or just numbers are used is up to you, although just numbers are often used in an ascending sequence (that is, 1, 2, 3, and so on). If you adopt this strategy, you could calculate the values by calling a stateless Session bean—a number fountain, if you will. A home method could perhaps encapsulate the lookup of this `NumberFountainBean`.

The implementation of such a `NumberFountainBean` can take many forms. There will need to be some persistent record of the maximum allocated number in the series, so a method such as `getNextNumber("MyBean")` could return a value by performing an appropriate piece of SQL against a table held in an RDBMS:

```
begin tran

update number_fountain
set    max_value = max_value + 1
where  bean_name = "MyBean"

select max_value
from   number_fountain
where  bean_name = "MyBean"

commit
```

One disadvantage with this approach is that the `NumberFountainBean`—or rather, the underlying database table—can become a bottleneck. A number of strategies have been developed to reduce this. One is to make the `getNextNumber()` method occur in a different transaction from the rest of the work. You will learn more about transactions on Day 8; for now, it is just necessary to know that while this will increase throughput, there is the risk of gaps occurring in the sequence.

If non-contiguous sequences are acceptable, even better throughput can be achieved by implementing a stateless Session bean that caches values in memory. Thus, rather than incrementing the maximum value by 1, it can increment by a larger number, perhaps by 100. Only every 100th call actually performs an SQL update, and the other 99 times, the number is allocated from memory. Of course, if the system crashes or power fails, there could be quite a large gap.

A final enhancement that improves scalability further and also improves resilience is to arrange for there to be a number of beans, each with a range of values. For example, these might be allocated using a high-order byte/low-order bytes arrangement.

Countries that allocate car license plates by state or by district are effectively using this approach.

One advantage of implementing a Session bean, such as `NumberFountainBean`, is that it isolates the dependency on the persistent data store that is holding the maximum value. Also, the SQL to determine the next available number is easily ported across RDBMS. On the other hand, many organizations use only a single RDBMS, so such portability is not needed. In these cases, the RDBMS may have built-in support for allocating monotonically increasing numeric values, and this can be used directly. For example, `SEQUENCE`s can be used in Oracle, while both Microsoft SQL Server and Sybase provide so-called identity columns and the `@@identity` global variable. So, for BMP Entity beans, another way to obtain the next value is to perform the SQL `INSERT`, obtaining the value from the back-end RDBMS. Note that most of these tools have the same scalability issues as the home-grown `NumberFountainBean`, and most also provide optimizations that can result in gaps in the series.

There is an alternative to using numbers for ID values, namely to generate a value that must be unique. On Windows PCs, you may well have seen strings in the format

```
{32F8CA14-087C-4908-B7C4-6757FE7E90AB}
```

In case you are wondering, this was found by delving into the Windows Registry and (apparently) represents the `FormatGUID` for .AVI files (whatever that means!). The point

is that it is—to all intents and purposes—guaranteed to be unique. In the case of GUIDs, it is unique because it is based on the MAC address of the ethernet card of the PC, plus the time.

Clearly, other algorithms can be created, and a quick search on the Web should throw up some commercial products and free software from which to select. For example, one algorithm generates values unique to the millisecond, the machine, the object creating the ID, and the top-most method of the call stack.

Granularity Revisited

A recurrent theme when developing Entity beans is in selecting an appropriate granularity for the bean. Prior to EJB 2.0, Entity beans could only provide a remote interface, which meant that a relatively coarse grained interface was required to minimize the client-to-bean network traffic. Indeed, this is still the recommendation made for Session beans in EJB 2.0 that have a remote interface.

With EJB 2.0, Entity beans can have a local interface, meaning that the cost of interaction with the client becomes minimal. If the cost of interaction of the Entity bean to the persistent data store is not too high, fine-grained Entity beans are quite possible. This may be true, either because the EJB container can optimize the database access in some way (true only for CMP Entity beans) or if the data store resides on the same computer as the EJB container and, ideally, within the same JVM process space.

> Running a persistent data store in the same process space as the EJB container is quite possible; a number of pure Java RDBMS—including Cloudscape, the database bundled with the J2EE RI—provide an "embedded mode."

Under BMP however, the advice is generally *not* to use fine-grained Entity beans, principally because the EJB container will be unable to perform any database access optimization. Choosing the granularity is then best determined by focusing on the identity and lifecycle on the candidate Entity beans. Hence, order and order-detail should be a single `Order` bean, but `customer` and `order`, while related, should be kept separate.

In the case study, you will find that the `Job` bean writes to both the `Job` table and also the `JobSkill` table (to record the skill(s) needed to perform the job).

Beware Those Finder Methods!

As you now have learned, Entity beans can be created, removed, and found through their home interface. While these all seem straightforward enough operations, there's danger

lurking in the last of these; finder methods can cripple the performance of your application if used incorrectly.

This probably doesn't seem obvious, but if you consider the interplay between the EJB container (implementing the local-home interface) and the bean itself, it becomes easier to see:

- The local-home interface's `findManyBy`*`Xxx`*`()` method is called. For the purposes of this discussion, this finder method returns a `Collection`.
- The local-home interface delegates to a pooled bean, calling its `ejbFindManyByXxx()` method. This performs an SQL `SELECT` statement (or the equivalent), and returns back a `Collection` of primary keys to the local-home interface.
- The local-home interface instantiates a `Collection` the same size as was obtained from the bean and populates it with local proxies. Each local proxy is assigned a primary key.

So far so good, the client receives a `Collection` of proxies. Suppose now that the client iterates over this collection, calling some getter method `get`*`Xxx`*`()`.

- The client calls `get`*`Xxx`*`()` on the first proxy in its `Collection`. The proxy holds a primary key, but there is no corresponding bean actually associated with the proxy. Therefore, the EJB container activates a bean from the pool, calls its `ejbLoad()` lifecycle method, and then finally delegates the `get`*`Xxx`*`()` business method. After that method has completed, the `ejbStore()` method is called.
- This process continues for all of the proxies in the collection.

You can probably see the problem; the persistent data store will be hit numerous times. First, it will be hit as a result of the `ejbFindManyBy`*`Xxx`*`()` method; this will return a thin column of primary key values. Then, because `ejbLoad()` is called for each bean, the rest of the data for that row is returned. This is shown in Figure 6.8.

Consequently, if 20 primary keys were returned by the bean following the initial `ejbFindManyBy`*`Xxx`*`()`, the network would be transversed 21 times, and the database will be hit in all 41 times—once for the initial `SELECT` and two times each for each of the beans.

There are a number of techniques that can eliminate this overhead, each with pros and cons:

- The most obvious solution is to not use finder methods that return `Collection`s of many beans. Instead, use stateless Session beans that perform a direct SQL `SELECT` query against the database, iterate over the `ResultSet`, and return back a `Collection` of serializable value objects that mirror the data contained in the actual entity. This technique is called the *Fast-lane Reader*.

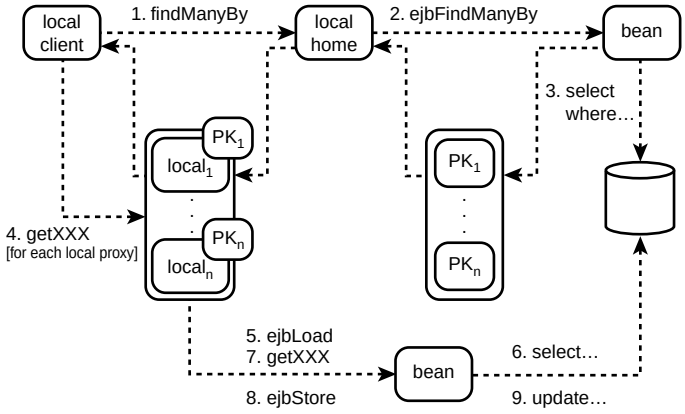


FIGURE 6.8

Finder methods can result in poor performance under BMP.

- Another technique that can be used is to alter the definition of the primary key class. As well as holding the key information that identifies the bean in the database, it also holds the rest of the bean's data as well. When the original finder bean returns the `Collection` of primary keys, the primary keys are held by the local proxies. When the beans are activated, they can obtain their state from the proxy by using `entityContext.getLocalObject().getPrimaryKey()`. This technique has been dubbed the *fat key* pattern, for obvious reasons.
- Last, you may be able to remove the performance hit by porting the bean to use container managed persistence. Because under CMP the EJB container is responsible for all access to the persistent data store, many will obtain all the required information from the data store during the first `findManyBy`*`Xxx`* method call, and then eagerly instantiate beans using this information. You will be learning more about CMP tomorrow.

Incidentally, Figure 6.8 shows a graphic illustration of why Entity beans should, in general, define only a local interface—not a remote interface. If the client were remote rather than local, the total network calls for a finder method returning references to 20 beans would be double the original figure, namely 42! Moreover, every subsequent business method invocation (call to `get`*`YYY`*`()` for example) would inflict a further 20 network calls.

EJB Container Performance Tuning

Many organizations are wary of using Entity beans because of the performance costs that are associated with it. You have already seen the performance issues arising from using finder methods, but even ignoring this, any business method to an Entity bean will require two database calls—one resulting from the `ejbLoad()` that precedes the business method and one from the `ejbStore()` to save the bean's new state back to the data store.

Of course, these database calls may be unnecessary. If a bean hasn't been passivated since the last business call, the `ejbLoad()` need not do anything, provided that nothing has updated the data store through non-EJB mechanisms. Also, if the business method called did not change the state of the bean the `ejbStore()` has nothing to do also.

Another scenario is where a bean is interacted with several times as part of a transaction. You will be learning more about transactions on Day 8, so for now, just appreciate that when a bean is modified through the course of a transaction, either all of its changes in state or none of them need to be persisted. In other words, there is only really the requirement to call `ejbStore()` just once at the end of the transaction.

Taking these points together, the amount of network traffic from the EJB container to the persistent data store can be substantially reduced, down to the levels that might be expected in a hand-written J2SE client/server application. Although not part of the EJB specification, many EJB containers provide proprietary mechanisms to prevent unnecessary `ejbLoad()` or `ejbStore()` calls. Of course, the use of these mechanisms will make your bean harder to port to another EJB container, but you may well put up with the inconvenience for the performance gains realized. Indeed, if you are in the process of evaluating EJB containers, as many companies are, you may even have placed these features on your requirements list.

Configuring and Deploying a BMP Entity Bean

Yesterday, you learned how to deploy Session beans by creating `ejb-jar` files with their own deployment descriptor and including the `ejb-jar` into an enterprise application. Deploying Entity beans is done in precisely the same way, by creating `ejb-jar` files that contain the Entity bean classes, with appropriate entries in the deployment descriptor. This deployment descriptor is the same deployment descriptor as for Session beans. As you saw yesterday, the root element of this deployment descriptor is the `ejb-jar` element:

```
<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?,
enterprise-beans, relationships?, assembly-descriptor?, ejb-client-jar?)>
```

Looking at the `enterprise-beans` element, this is defined as follows:

```
<!ELEMENT enterprise-beans (session | entity | message-driven)+>
```

The deployment descriptor can contain Session, Entity, and/or Message-driven beans. Or put another way, Session beans and Entity beans can be placed in the same `ejb-jar` if required.

Entity Element

The `entity` element of the DTD is defined as follows:

```
<!ELEMENT entity (description?, display-name?, small-icon?, large-icon?,
ejb-name, home?, remote?, local-home?, local?, ejb-class,
persistence-type,
prim-key-class,
reentrant,
cmp-version?, abstract-schema-name?,
cmp-field*, primkey-field?,
env-entry*,
ejb-ref*, ejb-local-ref*,
security-role-ref*, security-identity?,
resource-ref*,
resource-env-ref*,
query*)>
```

Many of the elements referred by the `entity` element are also used by the `session` element, and were discussed yesterday:

- *Presentational elements*—`description`, `display-name`, `small-icon`, and `large-icon`.
- *Elements describing the components of the bean*—`ejb-name`, `home`, `remote`, `local-home`, `local`, and `ejb-class` elements. The `local-home` and `local` elements just name the interfaces that extend `javax.ejb.EJBLocalHome` and `javax.ejb.EJBLocal`, respectively.
- *Elements referring to the environment and other resources*—`env-entry`, `ejb-ref`, `ejb-local-ref`, `resource-ref`, and `resource-env-ref`.

The remaining elements are specific to Entity beans:

- `persistence-type` Set to `Bean` for bean-managed persistence or `Container` for container-managed persistence. For the purposes of today, this will be set to `Bean`.
- `prim-key-class` This mandatory element indicates the name of the primary key class. This will be a custom developed class for beans that have primary keys (`JobPK` for `Job` bean) but may be a regular class (`java.lang.String`) for others (for example, the `Location` bean).
- `cmp-version`, `abstract-schema-name`, `cmp-field`, `prim-key-field`, `query` These are used only for CMP beans and are covered tomorrow.
- `reentrant` Set to `true` for reentrant Entity beans, or `false` for non reentrant beans. It's safer to mark Entity beans as non reentrant.
- `security-role-ref`, `security-identity` You will learn more about security on Day 15.

Listing 6.12 shows the deployment descriptor for the `Job` bean.

LISTING 6.12 `entity` Element Descriptor for the `Job` Bean

```
 1: <entity>
 2:     <display-name>JobBean</display-name>
 3:     <ejb-name>JobBean</ejb-name>
 4:     <local-home>data.JobLocalHome</local-home>
 5:     <local>data.JobLocal</local>
 6:     <ejb-class>data.JobBean</ejb-class>
 7:     <persistence-type>Bean</persistence-type>
 8:     <prim-key-class>data.JobPK</prim-key-class>
 9:     <reentrant>False</reentrant>
10:     <ejb-local-ref>
11:         <ejb-ref-name>ejb/SkillLocal</ejb-ref-name>
12:         <ejb-ref-type>Entity</ejb-ref-type>
13:         <local-home>data.SkillLocalHome</local-home>
14:         <local>data.SkillLocal</local>
15:         <ejb-link>data_entity_ejbs.jar#SkillBean</ejb-link>
16:     </ejb-local-ref>
17:     <ejb-local-ref>
18:         <ejb-ref-name>ejb/LocationLocal</ejb-ref-name>
19:         <ejb-ref-type>Entity</ejb-ref-type>
20:         <local-home>data.LocationLocalHome</local-home>
21:         <local>data.LocationLocal</local>
22:         <ejb-link>data_entity_ejbs.jar#LocationBean</ejb-link>
23:     </ejb-local-ref>
24:     <ejb-local-ref>
25:         <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
26:         <ejb-ref-type>Entity</ejb-ref-type>
27:         <local-home>data.CustomerLocalHome</local-home>
28:         <local>data.CustomerLocal</local>
29:         <ejb-link>data_entity_ejbs.jar#CustomerBean</ejb-link>
30:     </ejb-local-ref>
31:     <security-identity>
32:     <description></description>
33:     <use-caller-identity></use-caller-identity>
34:     </security-identity>
35:     <resource-ref>
36:         <res-ref-name>jdbc/Agency</res-ref-name>
37:         <res-type>javax.sql.DataSource</res-type>
38:         <res-auth>Container</res-auth>
39:         <res-sharing-scope>Shareable</res-sharing-scope>
40:     </resource-ref>
41: </entity>
```

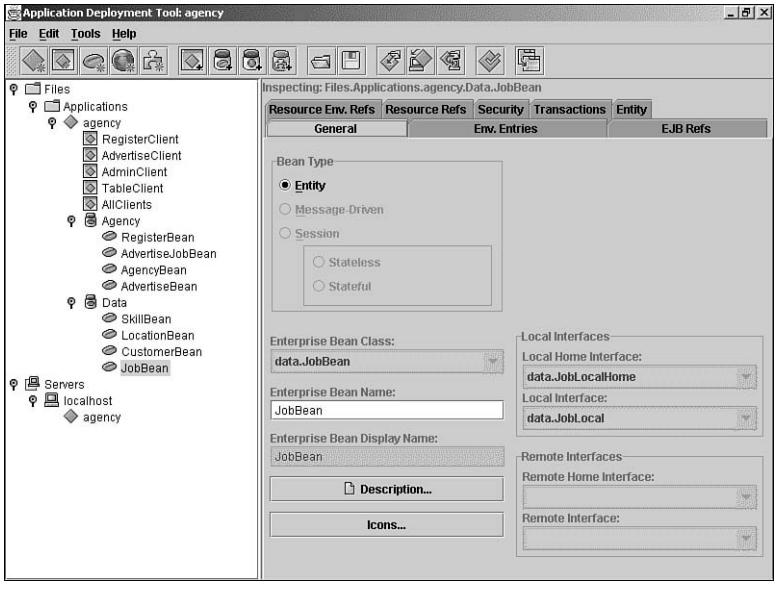
Note the `resource-ref` dependency on `jdbc/Agency`, just as you saw for Session beans that make JDBC calls. The `res-ref-name` is the coded name `jdbc/Agency` that appears in the `setEntityContext()` method of `LocationBean`. This logical reference is mapped

to the physical resource through the auxiliary deployment descriptor `agency_ea-sun-j2ee-ri.xml`. There are also `ejb-ref` dependencies on various other beans; these are ultimately used to manage the relationships with the `Location`, `Skill`, and `Customer` beans.

All of this information can be seen through the J2EE RI `deploytool` GUI, as shown in Figure 6.9.

FIGURE 6.9

The `deploytool` GUI displays deployment descriptor information graphically.



Good though the `deploytool` GUI is, sometimes a command-line interface is required. For example, you might want to automatically compile and then re-deploy your enterprise application in a project as part of a nightly build. Luckily, the `deploytool` also provides a command-line interface, so today you will deploy the Entity beans using this mechanism.

Under `day06\agency`, the directory structure has been organized into a number of subdirectories, as shown in Table 6.2.

TABLE 6.2 Directory Structure under `day06\agency`

Subdirectory	*Contents*
`build`	a) `buildAll`, which calls most of the other scripts in this directory, except b) `deploy`, which deploys the enterprise application.
`src`	Java source for the `client`, `agency`, and `data` packages.

TABLE 6.2 Continued

Subdirectory	*Contents*
`dd`	Standard XML deployment descriptors for the EJBs, for the enterprise application, and for the application clients. This directory also contains auxiliary deployment descriptors used by the J2EE RI to map the logical references in the standard deployment descriptors to the physical runtime environment managed by the J2EE RI container.
`classes`	Compiled Java classes. The various `compile*` scripts in the `build` directory compile into this directory.
`jar`	`ejb-jar` and `ear` (Enterprise Application) archives. Generated by the various `build*` scripts in the `build` directory.
`run`	The scripts to run the various clients.

To deploy the enterprise application, start Cloudscape and the J2EE RI, and then type

```
> cd day06\agency\build
> buildAll
> deploy
```

It's as simple as that!

One of the benefits of this approach is that bean developers, application assemblers and deployers who are familiar with the standard EJB deployment descriptors can modify the bean's deployment configuration by simply editing the XML deployment descriptors directly—without having to get to grips with the J2EE RI GUI. Moreover, some of the names automatically assigned by the `deploytool` GUI, such as the names of the `ejb-jar` files themselves, can be given more descriptive names. Consequently, the `ejb-jar` that contains the Entity beans is called `data_entity_ejbs.jar` and its deployment descriptor is called `data_entity_ejbs_ejb-jar.xml`.

If you look in the `dd` directory (or in the `deploytool` GUI as shown in Figure 6.9), you can see that the case study separates out the Session beans and the Entity beans into two different `ejb-jar`s. The `Agency ejb-jar` contains the same set of Session beans that you saw yesterday (although their implementation is different as you shall see shortly, they now delegate to the Entity beans), while the `Data ejb-jar` has the new BMP Entity beans. How you choose to organize your enterprise application is up to you.

Client's View

You've now learned how to specify, implement, and deploy BMP Entity beans, but how are they used? As you can probably imagine, the steps to obtain a reference to an Entity bean are similar to that for Session beans:

1. Look up the home interface for the Entity bean from JNDI.
2. To create a new entity instance, use the relevant `home.create()` method.
3. To locate an existing entity instance, use `home.findByPrimaryKey()` if the primary key is known, or some other `home.find`*`Xxx`*`()` finder method to obtain a `Collection` of matching entities.
4. For the returned local proxy to the Entity bean, invoke the business methods defined in the local interface of the bean.

The `javax.ejb.EJBLocalObject` interface also defines a number of other methods that can be called by the client, and it is worth discussing the semantics of these briefly:

- The `getPrimaryKey()` method of `EJBLocalObject` returns the primary key that identifies the bean.

> Note that because `EJBLocalObject` is also the super-interface for Session bean interfaces, this method can also be called when the client has a reference to the local proxy of a Session bean. However, because primary keys do not make sense for Session beans, an `EJBException` will always be thrown.

- If an Entity bean has both a local and a remote interface, and then `EJBObject.getPrimaryKey()` (from the remote proxy) and `EJBLocalObject.getPrimaryKey()` (from the local proxy) will both return objects that are equal (according to the definition the primary key's definition of `equals()`).
- The `isIdentical()` method can be used instead of comparing primary key classes to determine if two bean references refer to the same Entity bean. In other words, `bean1.isIdentical(bean2)` returns `true` if and only if `bean1.getPrimaryKey().equals(bean2.getPrimaryKey())`.

One scenario that can occur is that a client can have a reference to an Entity bean, and then the bean could be deleted by some other client. This could occur either by an EJB application client that invokes `remove()` on the same Entity bean, or it could be a non-EJB client that deletes the data directly from the underlying persistent data store. Either way, the original client will not be notified of this, and won't detect this situation until it next invokes a method on the Entity bean. In this case, the client will receive a `javax.ejb.NoSuchObjectLocalException` (a subclass of `javax.ejb.EJBException`, in turn a subclass of `java.lang.RuntimeException`) if accessing the Entity bean through its local interface.

The `javax.ejb.NoSuchObjectLocalException` caught by local clients is analogous to the `java.rmi.NoSuchObjectException` that would be caught if accessing the Entity bean through its remote interface. Table 6.3 shows the table from yesterday detailing various other exceptions, supplemented with the exception classes received by local clients.

TABLE 6.3 System Exceptions Are Thrown in a Variety of Situations

What	*Event*	*Local Client Receives*	*Remote Client Receives*
Any bean	Throws `javax.ejb.EJBException` (or any subclass)	`javax.ejb.EJBException` (or subclass)	`java.rmi.RemoteException`
BMP Entity bean	Throws `NoSuchEntityException`	`javax.ejb.NoSuchEntityException`	`java.rmi.NoSuchObjectException`
Container	When client invokes method on a reference to a bean that no longer exists	`javax.ejb.NoSuchObjectLocalException`	`java.rmi.NoSuchObjectException`
	When client calls a method without a transaction context	`javax.ejb.TransactionRequiredLocalException`	`javax.transaction.TransactionRequiredException`
	When client has insufficient security access	`javax.ejb.AccessLocalException`	`java.rmi.AccessException`
	When transaction needs to be rolled back	`javax.ejb.TransactionRolledBack LocalException`	`javax.transaction.TransactionRolledBack Exception`

As you can see, the EJB Specification makes some attempt at a naming standard so that the models are as similar as possible for local and remote clients.

Session Beans Revisited

The case study for today has the same set of Session beans as yesterday, and their interfaces are the same. However, their implementation is quite different, because they delegate all database interactions to the Entity bean layer.

As an example, Listing 6.13 shows the original `updateDetails()` method in the stateful `AdvertiseJob` bean. The `AdvertiseJob` bean provides services for managing jobs.

LISTING 6.13 `AdvertiseJobBean.updateDetails()` Without an Entity Bean Layer

```
 1: package agency;
 2:
 3: import java.util.*;
 4: import javax.ejb.*;
 5: import java.sql.*;
 6: // imports omitted
 7:
 8: public class AdvertiseJobBean extends SessionBean
 9: {
10:     public void updateDetails(String description,
                ➥String location, String[] skills) {
11:         if (skills == null) {
12:             skills = new String[0];
13:         }
14:         Connection con = null;
15:         PreparedStatement stmt = null;
16:         try {
17:             con = dataSource.getConnection();
18:             stmt = con.prepareStatement(
                ➥"UPDATE JOB
                ➥ SET description = ?, location = ?
                ➥ WHERE ref = ? AND customer = ?");
19:             stmt.setString(1, description);
20:             stmt.setString(2, location);
21:             stmt.setString(3, ref);
22:             stmt.setString(4, customer);
23:             stmt.executeUpdate();
24:             stmt = con.prepareStatement(
                ➥"DELETE FROM JobSkill
                ➥ WHERE job = ? AND customer = ?");
25:             stmt.setString(1, ref);
26:             stmt.setString(2, customer);
27:             stmt.executeUpdate();
28:             stmt = con.prepareStatement(
                ➥"INSERT INTO JobSkill (job, customer, skill)
                ➥ VALUES (?, ?, ?)");
29:             for (int i = 0; i < skills.length; i++) {
30:                 stmt.setString(1, ref);
31:                 stmt.setString(2, customer);
32:                 stmt.setString(3, skills[i]);
33:                 stmt.executeUpdate();
34:             }
35:             this.description = description;
36:             this.location = location;
```

LISTING 6.13 Continued

```
37:             this.skills.clear();
38:             for (int i = 0; i < skills.length; i++)
39:                 this.skills.add(skills[i]);
40:         }
41:         catch (SQLException e) {
42:             error("Error updating job " + ref + " for " + customer, e);
43:         }
44:         finally {
45:             closeConnection(con, stmt, null);
46:         }
47:     }
48: }
```

Listing 6.14 shows the updated version, delegating the hard work to the `Job` bean:

LISTING 6.14 `AdvertiseJobBean.updateDetails()` with an Entity Bean Layer

```
 1: package agency;
 2:
 3: import java.util.*;
 4: import javax.ejb.*;
 5: import data.*;
 6: // imports omitted
 7:
 8: public class AdvertiseJobBean extends SessionBean
 9: {
10:     private JobLocal job;
11:     public void updateDetails(String description,
                ➥String locationName, String[] skillNames) {
12:         if (skillNames == null) {
13:             skillNames = new String[0];
14:         }
15:         List skillList;
16:         try {
17:             skillList = skillHome.lookup(Arrays.asList(skillNames));
18:         } catch (FinderException ex) {
19:             error("Invalid skill", ex); // throws an exception
20:             return;
21:         }
22:         LocationLocal location = null;
23:         if (locationName != null) {
24:             try {
25:                 location = locationHome.findByPrimaryKey(locationName);
26:             } catch (FinderException ex) {
27:                 error("Invalid location", ex); // throws an exception
28:                 return;
29:             }
30:         }
```

LISTING 6.14 Continued

```
31:         job.setDescription(description);
32:         job.setLocation(location);
33:         job.setSkills(skillList);
34:     }
35:     // code omitted
36: }
```

The updated version is much more object-oriented; the knowledge of the database schema has been encapsulated where it rightfully belongs—in the Entity bean layer.

All this means that the `AdvertiseJob` bean no longer has any dependencies on the `jdbc/Agency DataSource`. On the other hand, it does now have dependencies on several of the Entity beans. These are defined using `ejb-local-ref` elements in the deployment descriptor. The relevant portion of the `AdvertiseJob` deployment descriptor (`agency_session_ejbs_ejb-jar.xml` file in the `dd` directory) is shown in Listing 6.15:

LISTING 6.15 `AdvertiseJob` Bean's Reference to the Entity Beans

```
 1: <ejb-local-ref>
 2:     <ejb-ref-name>ejb/SkillLocal</ejb-ref-name>
 3:     <ejb-ref-type>Entity</ejb-ref-type>
 4:     <local-home>data.SkillLocalHome</local-home>
 5:     <local>data.SkillLocal</local>
 6:     <ejb-link>data_entity_ejbs.jar#SkillBean</ejb-link>
 7: </ejb-local-ref>
 8: <ejb-local-ref>
 9:     <ejb-ref-name>ejb/LocationLocal</ejb-ref-name>
10:     <ejb-ref-type>Entity</ejb-ref-type>
11:     <local-home>data.LocationLocalHome</local-home>
12:     <local>data.LocationLocal</local>
13:     <ejb-link>data_entity_ejbs.jar#LocationBean</ejb-link>
14: </ejb-local-ref>
15: <ejb-local-ref>
16:     <ejb-ref-name>ejb/JobLocal</ejb-ref-name>
17:     <ejb-ref-type>Entity</ejb-ref-type>
18:     <local-home>data.JobLocalHome</local-home>
19:     <local>data.JobLocal</local>
20:     <ejb-link>data_entity_ejbs.jar#JobBean</ejb-link>
21: </ejb-local-ref>
22: <ejb-local-ref>
23:     <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
24:     <ejb-ref-type>Entity</ejb-ref-type>
25:     <local-home>data.CustomerLocalHome</local-home>
26:     <local>data.JobLocal</local>
27:     <ejb-link>data_entity_ejbs.jar#CustomerBean</ejb-link>
28: </ejb-local-ref>
```

Note the `ejb-link` reference, which names the bean that implements the required interfaces. This notation is used rather than a JNDI name because JNDI can (potentially) refer to remote EJBs, whereas local EJBs must—by definition—be deployed on the same server.

Patterns and Idioms

Yesterday, you saw some patterns and idioms that apply to writing Session beans. In this section, you will see some more that relate to BMP Entity beans and local interfaces. If you are impatient to get onto the exercise, feel free to skip this section and revisit later.

Interfaces, Façades, and State

This is a pattern that relates mostly to Session beans, but is discussed today rather than yesterday because it uses local interfaces.

While Session beans can provide both a remote and a local interface, you'll only want to provide one or the other more often than not. Generally, you should aim to have a small number of Session beans that offer a remote interface, and the remainder will be local "helper" beans. In design pattern terms, the remote Session beans create a Façade. This pattern is discussed more fully on Day 18, "Patterns."

The Façade beans will likely be stateful. Certainly, the helper beans should not be stateful, because chaining stateful objects is poor design, as noted yesterday.

Use Local Interfaces for Entity Beans

Entity beans should only ever be accessed through a local interface, and there are some very good reasons for this.

First, accessing Entity beans through a remote interface implies network traffic and the cost of cloning serializable objects. When the client runs in the same JVM as the Entity bean, a local interface eliminates cost.

This ties in with the previous discussion on Session bean interfaces and façades. Session beans provide a remote interface to the enterprise application, so Session beans should act as a front-end to Entity beans. Some people think of this as the *verb/noun* paradigm—the Session beans are the verbs (the doing end of the application) and the Entity beans are the nouns (the reusable business objects within some domain). Defining only local interfaces to Entity beans effectively enforces this pattern.

Second, finder methods of Entity beans—already expensive to use—become even more so when the client is remote. This was remarked on earlier today.

Lastly, and perhaps most significantly, local interfaces are the cornerstone of container-managed relationships (CMR)—part and parcel of container-managed persistence. You'll be learning all about this tomorrow. If you build your BMP beans using local interfaces, you provide a migration path to implementing those beans using CMP in the future.

Dependent Value Classes

Under BMP, the Entity bean's state can be represented in any way that is appropriate. Moreover, the bean can persist this state, in any way it wants.

Sometimes, a bean's state will be simple enough (that is, just a set of scalar fields) that it will correspond to a row in an RDBMS table. More often though, some of those fields will be vector, and they might even include some complex data, such as a map or photo and so on.

One simple solution to persisting such objects is to ensure that the fields of the bean are either scalar primitives or are a reference to a serializable object, an instance of what are sometimes called dependent value classes. The structure of that object can be as complex as needed, so long as it is serializable. The scalar fields are stored in regular columns in the RDBMS table, and the serializable object is stored as a binary large object (BLOB).

For example, the `Job` bean is mapped to both the `Job` and `JobSkill` table in the case study, accessing both in its `ejbLoad()` and `ejbStore()` methods. The `JobBean.skills` field is a `List` of `SkillLocal` references.

An alternative design would be to store the `skills` list as a BLOB in the database. The `Job` table would be redefined to be something like the following:

```
create table Job
(ref varchar(16),
 customer varchar(16),
 description varchar(512),
 location varchar(16),
 skills long varbinary       -- store a BLOB in Cloudscape
)
```

There is then a one-to-one mapping between an instance of the `Job` Entity bean and a row in the `Job` table. One slight complication is that because the `JobBean.skills` field contains references to `SkillLocal` references, which are not necessarily serializable, the `skills` variable would not be serializable either. So, in the `ejbStore()` method, a `List` of skill names (that is, primary key to each `SkillLocal`) would be created and saved to the database. This `List` of skill names is the "dependent value class."

```
stmt = con.prepareStatement(
➥"UPDATE Job
➥ SET description = ?, location = ?, skills = ?
➥ WHERE ref = ? AND customer = ?");
```

```
stmt.setString(1, description);
if (location != null) {
    stmt.setString(2, location.getName());
} else {
    stmt.setNull(2, java.sql.Types.VARCHAR);
}

List skillNameList = new ArrayList();
for(Iterator iter = this.skills.iterator(); ) {
    skillNameList.add( ( (SkillLocal)iter.next() ).getName() );
}
stmt.setBlob(3, skillNameList);

stmt.setString(4, ref);
stmt.setString(5, customerObj.getLogin());
stmt.executeUpdate();
```

Conversely, in the `ejbLoad()`, the `List` of skill names would be converted back to a `List` of `SkillLocal` references. The `SELECT` statement would be as follows:

```
stmt = con.prepareStatement(
"SELECT description,location,skills FROM Job WHERE ref = ? AND customer = ?");
```

and the processing of the result set would be

```
this.description = rs.getString(1);

String locationName = rs.getString(2);
this.location = (locationName != null)?
➥locationHome.findByPrimaryKey(locationName):null;

List skillNameList = (List)rs.getBlob(3);
this.skills =   skillHome.lookup(skillNameList);
```

The `skillHome.lookup()` home method of the `Skill` bean does the actual conversion from `name` to `SkillLocal`. (In fact, this method is actually used in `JobBean`'s `ejbLoad()` method, so you can check out the code yourself).

This approach can substantially reduce the coding effort, although you should also be aware of some of the downsides:

- First, the BLOB field is effectively atomic; even if just a small piece of information is changed (for example, a new skill is added), and then entire BLOB must be replaced.
- Furthermore, information previously easily accessible can now only be accessed through a single route. In the previous example, the data that was previously in the `JobSkill` table is now buried within the `Job.skills` field. It is no longer possible to perform an efficient SQL `SELECT` to find out which jobs require a certain skill. Instead, such a query will involve instantiating and then querying every `Job` bean instance.

- Last, the data in the persistent data store is stored in Java's own serializable format. While this is a well-defined structure, it nevertheless makes it non-trivial for non-Java clients to access this data.

It is possible to serialize Java classes in a custom-defined manner (for example, as an XML string) by either providing a `readObject()` and `writeObject()` method or by implementing `java.io.Externalizable` and implementing `readExternal()` and `writeExternal()` methods. However, such an approach would probably involve more development effort than had been saved by using a BLOB to store the dependent value class.

Because dependent value classes are serializable, it is also possible to use them as the return types of accessor methods (getters and setters) of the interface. Indeed, prior to the introduction of local interfaces in EJB 2.0, this was a recommended pattern to minimize network traffic across the Entity bean's remote interface. However, provided that only local interfaces are provided (and especially if using CMP), there is nothing wrong with providing fine-grained access to the values of the Entity bean. In effect, this design pattern has been deprecated with EJB 2.0.

Self-Encapsulate Fields

In the case study BMP beans, the private fields that represent state are accessed directly within methods. For example, the following is a fragment of the `JobBean.ejbCreate()` method:

```
public JobPK ejbCreate (String ref, String customer) throws CreateException {

    // database access code omitted

    this.ref = ref;
    this.customer = customer;
    this.description = description;
    this.location = null;
    this.skills = new ArrayList();

    // further code omitted
}
```

Some OO proponents argue that all access to fields should be through getter and setter accessor methods, even for other methods of the class. In other words, the principle of encapsulation should be applied everywhere. Using such an approach, the `ejbCreate()` method would be as follows:

```
public JobPK ejbCreate (String ref, String customer) throws CreateException {

    // database access code omitted

    setRef(ref);
    setCustomer(customer);
    setDescription(description);
    setLocation(null);
    setSkills(new ArrayList());

    // further code omitted
}
```

Some people find this overly dogmatic, and, indeed, the code in the case study takes the more direct approach. However, you may want to consider self-encapsulation because it makes BMP beans easier to convert to CMP. As you will see tomorrow, all accessing to the Entity bean's state must be through accessor methods.

Don't Use `Enumeration` for Finders

The EJB 1.0 specification was introduced before J2SE 1.2, so the specification allowed finder methods that returned many instances to return `java.util.Enumeration` instances. For backward compatibility, the EJB 2.0 specification still supports this, but you should always use `java.util.Collection` as the return type for finder methods returning more than one Entity bean instance.

Acquire Late, Release Early

In conventional J2SE programs, the idiom usually is to connect to the database at the start of the program when the user logs in, and only disconnect when the user logs out. Holding onto the open database connection while the user logs in substantially improves performance; database connections are relatively expensive to obtain. So, for J2SE programs, the mantra is "Acquire early, release late."

With J2EE programs, things are inverted. The database connection should be obtained just before it is required, and closed immediately after it has been used. In other words, "Acquire late, release early." This is shown in the `Job` bean, as shown in Listing 6.16.

LISTING 6.16 Acquire Late, Release Early, as Shown in `JobBean`

```
1: package data;
2:
3: import javax.ejb.*;
4: import java.sql.*;
5: // imports omitted
6:
```

LISTING 6.16 Continued

```
 7: public class JobBean implements EntityBean
 8:
 9:     public void ejbLoad() {
10:         JobPK key = (JobPK)ctx.getPrimaryKey();
11:         Connection con = null;
12:         PreparedStatement stmt = null;
13:         ResultSet rs = null;
14:         try {
15:             con = dataSource.getConnection();
16:             stmt = con.prepareStatement( … );
17:
18:             // SQL code omitted
19:
20:         }
21:         catch (SQLException e) {
22:             error("Error in ejbLoad for " + key, e);
23:         }
24:         catch (FinderException e) {
25:             error("Error in ejbLoad (invalid customer or location) for "
                    ➥ + key, e);
26:         }
27:         finally {
28:             closeConnection(con, stmt, rs);
29:         }
30:     }
31:     // code omitted
32: }
```

The reason that this works is because the database connection is obtained from a `javax.sql.DataSource` (line 15) in a J2EE environment, rather than using the `java.sql.DriverManager.getConnection()` method. Obtaining connections from `DataSources` is not expensive in performance terms because they are logical connections, not physical connections. When such a connection is obtained, it is merely obtained from a connection pool, and when it is "closed," it is simply returned back to the connection pool.

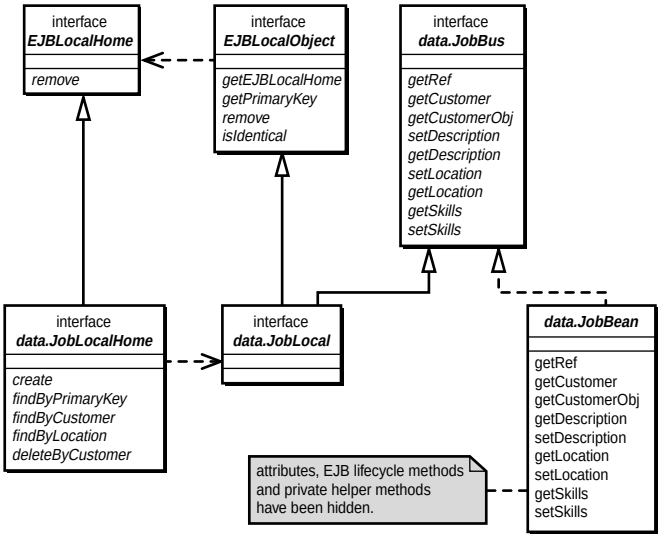
Indeed, using the J2SE idiom of acquire early, release late (for example, by obtaining a connection in `setEntityContext()` and releasing it in `unsetEntityContext()`) can adversely affect performance, because every instantiated bean would have its own database connection. This may well reduce application throughput because the memory resources of both the EJB container and the database server would be increased to handle many open database connections. In comparison, the J2EE idiom means that the number of database connections open is no more than the number of methods concurrently executing.

Business Interface Revisited

Yesterday, you learned about the business interface idiom, whereby the business methods are pulled out into a separate interface such that the bean itself can implement this interface. This principle can equally be applied to Entity beans using local interfaces, as shown in Figure 6.10.

FIGURE 6.10

Business interfaces can be applied to Entity beans with local interfaces.



There is one difference when applying this technique to beans that only have local interfaces; there is no longer any need for the methods of the business interface to throw `RemoteException`, because the local interface of the bean (`JobLocal` in Figure 6.10) is not itself remote. Even so, the bean still does not implement its local interface because the methods of the `EJBLocalObject` interface are there to be implemented by the local proxy object, not by the bean.

Gotchas

The following is a quick checklist of "gotchas" to help you with your implementation:

- Primary keys must be immutable. In other words, it is not possible to change the value of a primary key for an entity once assigned (see EJB specification, section 10.3.5).

  Of course, there is nothing to prevent you from directly changing the data in the underlying persistent data store (for example, with an SQL `UPDATE` statement). But you will need to do this with the EJB container offline or otherwise at rest.

- Sometimes, Entity beans interact with non-data store resources. An example might be a client `java.net.Socket` or perhaps a subscription to a JMS topic (covered more on Days 9, "Java Messaging Service," and Day 10, "Message-Driven Beans"). These resources will need to be acquired in both `ejbActivate()` (for an existing bean) and also for `ejbCreate()` (if the bean has just been created).

  Similarly, resources should be released in both `ejbPassivate()` and `ejbRemove()`. This is because a bean being deleted will not be passivated first.
- Finder methods can return `Collections`, but they can't return `Lists`, `Sets`, or `Maps`. However, this capability is planned for future versions of the EJB specification.
- If you have two bean references, note that the value of `bean1.equals(bean2)` is unspecified, and that `bean1 == bean2` is also unspecified. Moreover, `hashCode()` may differ for two references to the same underlying EJB. (All of these points are made in the EJB specification, section 9.8.)

  The correct way to compare bean identity is to use `bean.isIdentical()` or to use the `equals()` method on the primary key classes.
- Beware of a reliance on pass-by-reference side-effects when using local interfaces. Such a reliance would compromise portability.

Summary

Another long day, but you now have lots of good new material under your belt. You've learned that Entity beans represent persistent domain data with corresponding domain (not application) logic. You've seen that the constituent parts of Entity beans are pretty much the same as Session beans, though Entity beans also require a primary key class that must be custom-developed if the key is composite.

You've also learned that there are two different ways to implement Entity beans, either using bean-managed persistence, whereby the persistence code (JDBC, for example) resides within the bean code, or using container-managed persistence. You now know the lifecycle for BMP beans and how to implement such beans.

You saw that the EJB specification allows local interfaces to be defined for EJBs, as well as or instead of remote interfaces, and saw several good reasons why Entity beans should always use local interfaces.

Onto deployment, you now know that the J2EE RI allows EJBs can be deployed using a command line interface. A deeper understanding of the XML deployment descriptor is needed, but the process for deployment is (arguably) more portable and faster.

Finally, you've learned numerous design techniques, patterns, and idioms that should set you up for designing and implementing Entity beans effectively.

Q&A

Q What do Entity beans represent?

A Entity beans represent persistent data that can be accessed and shared by many clients over time.

Q What are the two types of Entity beans?

A The two types of Entity beans are BMP and CMP.

Q Why are local interfaces preferable to remote interfaces for Entity beans?

A Local interfaces perform better because there is no network traffic when calling a bean through its local interface, and there is also no need to clone serializable objects. They are also the basis for CMP.

Q How does a BMP Entity bean know what its primary key is?

A It can be passed as an argument of `ejbCreate()`, it could be generated by the RDBMS, it could be generated by some other bean, or it might be generated as a pseudo-random value using an algorithm that guarantees uniqueness.

Q Which two methods should the primary key class implement?

A The primary key class should implement the `hashCode()` and `equals()` methods.

Exercises

The exercise starts with a version of today's case study that has a complete set of Session beans, but an incomplete set of Entity beans. Where there is no Entity bean, the Session bean performs direct SQL. The state of affairs is shown in Table 6.4.

TABLE 6.4 Case Study Session and Entity Beans

Session Bean	*Functional Area*	*Functions*	*Implementation/ Delegation*
`Agency`	`Applicants`	create, delete, find all	Direct SQL
	`Customers`	create, delete, find all	`Customer` bean
	`Locations`	add, get details, get plural, remove	`Location` bean
	`Skills`	add, get details, get plural, remove	`Skill` bean
`Advertise`	`Job`	create, delete, get plural	`Job` bean
	`Customer`	get details, update	`Customer` bean
`AdvertiseJob`	`Job`	get details, update	`Skill` bean, `Location` bean
`Register`	`Applicant`	get details, update	Direct SQL

The exercise is to implement an `Applicant` Entity bean and to update the `Agency` and `Register` Session beans to use this new Entity bean.

The `Applicant` bean should map itself to the `Applicant` and `ApplicantSkill` tables and define the following fields:

- `login` This is the primary key for the `Applicant` Entity bean.
- `name` Simple scalar field.
- `email` Simple scalar field.
- `summary` Simple scalar field.
- `location` Should be a reference to a `LocationLocal` to ensure referential integrity.
- `skills` Should be a collection of `SkillLocal` references to ensure referential integrity.

You should find that the structure of your new bean shares many similarities with the `Job` Entity bean. One difference will be the primary key. The `Job` bean required a `JobPK` because it had a composite primary key. For your `Applicant` bean, you should not need to develop a custom primary key class because applicants will be identified simply by their `login`—a simple `String`.

The `ApplicantLocalHome` and `ApplicantLocal` interfaces have already been provided; note their similarity to `JobLocalHome` and `JobLocal`.

The directory structure of `day06\exercise` is the same as yesterday:

- `src` The source code for the EJBs and clients.
- `classes` Directory to hold the compiled classes; empty.
- `dd` Holds XML deployment descriptors.
- `build` Batch scripts (for Windows and UNIX) to compile the source and to build the EAR files into the `jar` directory.
- `jar` Holds `agency.ear`: the agency enterprise application. Also holds `agencyClient.jar`, the client-side JAR file optionally generated when deploy EAR. This directory also holds some intermediary JAR files that are used only to create the previous two `jar` files.
- `run` Batch scripts (for Windows and UNIX) to run the JARs. Use the files in the `jar` directory.

In the detailed steps that follow, note one difference from yesterday is that today you will be defining and configuring the EJB as part of the enterprise application by directly editing the XML deployment descriptors in the `dd` directory. If you feel uneasy about doing this, there is nothing to prevent you from making the changes through the GUI.

Do note, however, that the build scripts that create the `agency.ear` file do require that the `ApplicantBean.java` source exists (even if its implementation is incomplete).

The steps you should follow are:

1. Locate the `ApplicantBean.java` file within `day06\exercise\src\data`. This should have an empty implementation.
2. Implement `ApplicantBean` to support the `Applicant` and `ApplicantLocalHome` interfaces supplied. Base your implementation on `JobBean`, if you want.
3. Next, modify the `AgencyBean` Session bean. The `findAllApplicants()`, `createApplicant()`, and `deleteApplicant()` methods should instead delegate to `ApplicantHome`.
4. Now update the `RegisterBean` Session bean. In its `ejbCreate()` method, it should obtain a reference to an underlying `Applicant` Entity bean. Each of the business methods should then delegate to this applicant. If you want something to work from, look at the approach adopted by the `AdvertiseJob` Session bean, delegating to an instance of `Job` Entity bean.
5. Update the `data_entity_ejbs_ejb-jar.xml` deployment descriptor in the `dd` directory; again, cloning and adapting the `Job` bean entries will be a good start.
6. Update the `agency_session_ejbs_ejb-jar.xml` deployment descriptor to indicate the new dependencies of the `Agency` and `Register` Session beans. Both will depend on `ApplicantLocal`; you should also find that `Register` depends on `SkillLocal` and `LocationLocal` (to call the business methods of `Applicant`).
7. The `buildDataEntityEjbs` script already references `ApplicantBean`, so there is no need to change it. This causes your classes to be added to the resultant `data_entity_ejbs.jar` `ejb-jar` file.
8. Now, build the `jar\agency.ear` enterprise application by using `build\buildAll`. Load the resultant EAR file into `deploytool`, and check that the EJB is correctly defined. If it is not, either make the appropriate changes and run `buildAll` or make the changes through the `deploytool` GUI itself. Then, save the deployment descriptors into the `dd` directory.
9. Your `agency.ear` file is not quite ready to deploy, because the vendor-specific mapping information has not yet been specified. This is most easily generated by deploying the enterprise application from `deploytool`. The wizard that then appears will ensure that you have the opportunity to indicate any missing information. Then, test by using the `AllClients` client, invoked using the `run\runAll` script.

10. Optionally, you may want to save the auxiliary deployment descriptor to `dd\agency_ea-sun-j2ee-ri.xml`. If you do this, you will be able to build and deploy the application directly from the command line using `build\buildAll` and `build\deploy`, respectively. However, to obtain the auxiliary deployment descriptor, you will need to manually load the `agency.ear` file (from the previous step) into WinZip or equivalent and extract the auxiliary deployment descriptor; the `deploytool` GUI does not provide any direct mechanism.

Good luck. A working example can be found in `day06\agency` (with a correct auxiliary deployment descriptor).

WEEK 1

DAY 7

CMP and EJB QL

Yesterday, you learned how to specify, implement, and deploy bean-managed persistence (BMP) Entity beans. Today, you will learn

- How to specify, implement, configure and deploy CMP Entity beans
- How to use EJB Query Language (EJB QL)
- How to define relationships between CMP Entity beans

Overview of Container-Managed Persistence

The EJB specification provides for two different ways of implementing Entity beans. The first approach, covered yesterday, is for the bean provider to embed the persistence logic within the bean itself—hence the name bean-managed persistence or BMP. The second is for the container vendor to provide that logic, either generated by the EJB container vendor's deployment tools or as part of the EJB container itself. Entity beans built this way are called CMP Entity beans.

CMP Entity beans have always been part of the EJB specification, first in EJB 1.0 and then with some minor refinements in EJB 1.1. The changes to CMP Entity beans in EJB 2.0 are substantial—so substantial, in fact, that CMP 1.1 Entity beans are not forward compatible with EJB 2.0.

To deal with this, the EJB specification actually provides two different ways to write CMP Entity beans. The first is the legacy 1.1 approach; beans that are written this way indicate it using an entry in their deployment descriptor. The second is using the new and far more powerful approach introduced in EJB 2.0.

Today, you will be learning only about the new EJB 2.0 approach.

The "anatomy" of CMP Entity beans is very much the same as BMP Entity beans:

- They have a local-home (or remote home) interface that defines the create methods, the finder methods, optional home methods, and a remove method.
- They have a local (or remote) interface that defines the business methods of the bean.
- Obviously, they have the bean class itself that implements methods corresponding to the previously mentioned interfaces, and implements the lifecycle methods inherited from `javax.ejb.EntityBean`.
- Finally, they may have a primary key class (and must have one if the primary key is composite).

However, there are some differences. The responsibilities of the bean in the lifecycle methods are different, because there is no longer any requirement to persist the bean's state. This raises the question as to when the state is persisted by the container, because it could be done either before the lifecycle method is called or after. There are changes in the interactions between the container and the bean, as you will see.

Another significant difference is the finder methods. Under BMP, the bean provider writes the appropriate finder methods that interact with the persistent data store. Under CMP, the container will do this work, so there is no longer any need to implement the finder methods in the bean. However, the bean provider must still specify the nature of the query to be performed to obtain the correct data from the data store. This is done using *EJB Query Language* (EJB QL), appearing in the bean's deployment descriptor. EJB QL shares many similarities with ANSI SQL 92, so you should not have too many difficulties picking it up.

Just as tables in relational databases have relationships, so too do Entity beans. You saw this yesterday with the relationship between the `Job` bean, which had relationships with the `Skill`, `Location`, and `Customer` beans. Under BMP, the bean provider must write the code that maintains all of these relationships explicitly. If CMP is used, these relationships can be declaratively defined using *container-managed relationships*, or CMR. Again, the declarations of these associations are in the bean's deployment descriptor.

Relationships between Entity beans are intrinsically fine-grained. For example, a many-to-many relationship between `Job` and `Skill` (indicating which skills are needed for such-and-such a job) would involve dealing with many (`job,skill`) tuples in the case study stored in the `JobSkill` table. You know that Entity beans can have either a local or a remote interface, and that it's good practice to only ever interact with an Entity bean through its local interface because this reduces network traffic. Because the performance cost of maintaining a fine-grained relationship across the network would be too severe, the EJB specification requires that container-manager relationships between Entity beans are defined only through local interfaces. Indeed, one of the primary reasons for the introduction of local interfaces in the EJB specification was to make CMR feasible.

N-tier Architecture (Revisited Again) and CMP Fields

CMP has an impact on the n-tier architecture that you seen have on several previous days. Figure 7.1 shows an update of a figure that you saw yesterday.

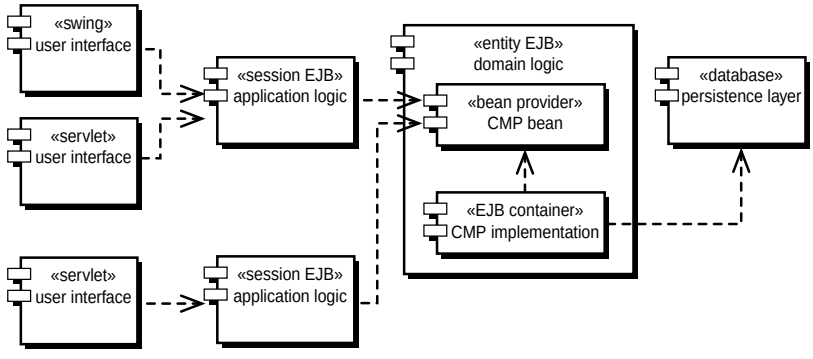


FIGURE 7.1

CMP Entity beans are split into two components.

There are still four tiers to the architecture—namely, the interface, application, domain, and persistence layers. However, with CMP, the Entity beans split into two components. The first component is provided by you, the bean provider. This defines the bean's local-home and local interfaces, but the implementation of the bean itself is incomplete.

It provides a full implementation of the business methods, but there is no implementation of the accessor methods for the bean's state. Indeed, you will see that the methods are marked as `abstract`. The concrete implementation of the CMP bean is completed by the EJB container provider. This component has dependencies on both the bean provider's bean, and—of course—on the persistence layer. The first dependency is because the concrete implementation uses the bean provider's `abstract` bean class as its superclass; in other words, it `extends` from the CMP bean. The second dependency is because the implementation of the bean performs appropriate data store calls.

You may recognize this design as an instance of the Template design pattern. The abstract CMP bean provided by the bean provider is a template, defining certain mandatory "hook" methods—namely, the accessor methods. The implementation of these hooks is provided by the EJB container in terms of the concrete CMP implementation.

Listing 7.1 shows this for the `Job` Entity bean. This bean defines a pair of accessor methods (the getter and setter) for each of its fields—`ref`, `customer`, `description`, `location`, and `skills`.

LISTING 7.1 The `JobBean`'s Fields Are Implied by the Presence of These Abstract Accessor Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: // imports omitted
 5:
 6: public abstract class JobBean implements EntityBean {
 7:
 8:     public abstract void setRef(String ref);
 9:     public abstract String getRef();
10:
11:     public abstract void setCustomer(String customer);
12:     public abstract String getCustomer();
13:
14:     public abstract String getDescription();
15:     public abstract void setDescription(String description);
16:
17:     public abstract LocationLocal getLocation();
18:     public abstract void setLocation(LocationLocal location);
19:
20:     public abstract Collection getSkills();
```

LISTING 7.1 Continued

```
21:     public abstract void setSkills(Collection skills);
22:
23:     // code omitted
24: }
```

Each of these accessors is implemented by the concrete CMP implementation. The actual instance variables are effectively part of the subclass' implementation, ultimately populated from the persistent data store.

This naming scheme throws up a very curious restriction, specifically that `cmp-fields` must not start with a capital letter. Thus, `customer` and even `cUSTOMER` are valid names, but `CUSTOMER` would not be. This is because the methods capitalize the `cmp-fields`, and one cannot capitalize a capital!

Another way of thinking about this design is in terms of vertical delegation. The Session beans can call the accessor methods on methods defined in the superclass, but the actual method that is invoked is the implementation defined in the subclass. The superclass CMP bean "delegates" vertically down to its subclass CMP implementation.

This design gives several advantages. One immediate advantage of this pattern is that it gives the EJB container (through the concrete bean implementation) much more control over populating the bean's state, without compromising good OO principles. For example, it is up to the EJB container whether it chooses to obtain all of the bean's state when the bean is activated or created (eager loading), or whether it chooses to fetch the bean's state from the persistent data store as and when needed (lazy loading). Indeed, the concrete implementation may adopt some half-way house, eagerly loading all the scalar data pertaining to the bean but lazily loading data corresponding to bean relationships. Indeed, this advantage is pointed out in the EJB specification (section 10.4.2.1).

Another advantage is that the EJB container only need persist the bean's state to the data store when the bean's state has changed. The concrete implementation can keep track of the before-and-after versions of the bean's state and compare them to see if any have changed as the result of a business method invocation. If a read-only accessor method (a "getter" method) is called, there would be no change in state and so the concrete implementation need not perform an unnecessary update to the data store. Taking this on one further stage, when the bean's state is changed, the EJB container need only update those fields that have changed and can ignore fields that have not changed. This reduces network traffic between the EJB container and the persistent data store.

One final advantage worth mentioning is that some value-add services, such as optimistic locking, can be implemented by EJB container vendors more straight-forwardly.

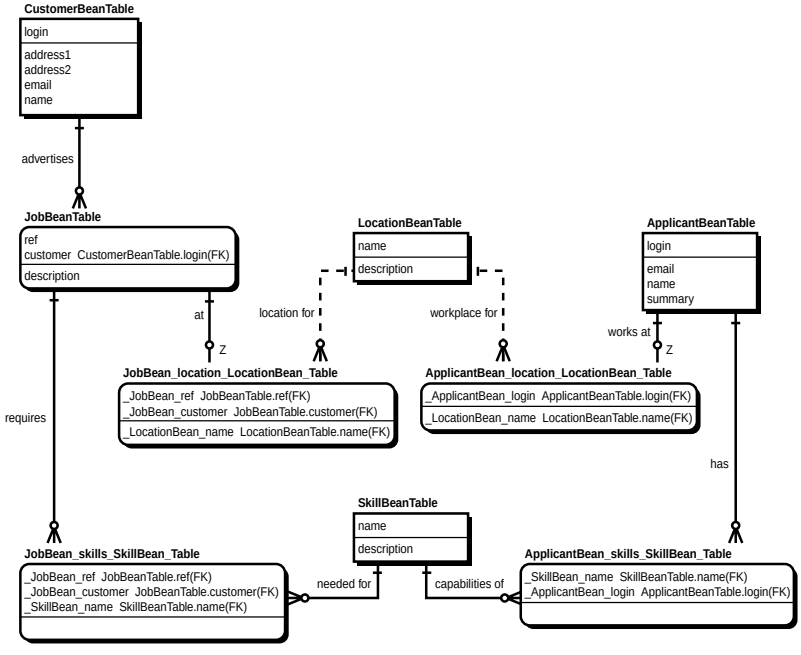
A Quick Word about the Case Study Database

As you go through the remaining topics for today, you may well want to load up and run the case study code. Before you can do this, the `Agency` database needs to be modified to support CMP. This is because the J2EE RI container generates its own SQL schema to store the Entity beans' data.

Figure 7.2 shows the revised schema for the case study, as generated by the J2EE RI container.

FIGURE 7.2

The case study database schema changes under CMP.



The J2EE RI container can automatically create this schema when the Entity beans are deployed. However, because there is example data in the case study database, it is easier to run the provided utility to "upgrade" the database to support CMP. The `Agency` Session bean queries the Cloudscape RDBMS tables directly. To ensure that this continues to work, the utility also creates SQL views with the names of the old tables (`JobSkill` and so on) against the new tables.

The steps for converting the database to support CMP are as follows:

1. Shut down Cloudscape if it is running.
2. Back up the current (pre day 7) version of the Agency database, under `%J2EE_HOME%\cloudscape`.

   Under Windows, you can do this using copy and paste, or from the command line type:

   ```
   > cd %J2EE_HOME%\cloudscape
   > xcopy Agency bmpAgency /I /E
   ```

   Under Unix, you can do this using the following:

   ```
   $ cd $J2EE_HOME/cloudscape
   $ cp –r Agency bmpAgency
   ```

3. Restart Cloudscape.
4. Under the case study `day07\Database` directory, run the batch `CreateCMPAgency.bat` (Windows) or `CreateCMPAgency.sh` (Unix). This calls `CreateCMPAgency.java` (already compiled for you) which, in turn, creates the previous tables, populates the tables with the same sample data, and creates the views for backward compatibility.
5. When you have done this, you may want to shut down Cloudscape and then back-up the CMP version of the database to a directory called `cmpAgency`, using similar commands to those in step 2. That way, you can easily switch between the two different schemas. To reinstate either version, just delete the `Agency` directory and then copy back either the `bmpAgency` or `cmpAgency` using `xcopy` (Windows) or `cp -r` (Unix).

CMP Entity Bean Lifecycle

The lifecycle for CMP Entity beans is substantially the same as BMP Entity beans, reflected in an almost identical diagram (see Figure 7.3).

This diagram differs from the lifecycle you saw yesterday in that there are no `ejbFindXxx()` methods for pooled beans. This is not to say that pooled CMP Entity beans do not perform finder methods; they do. However, the EJB container generates the actual code that performs this. There will be a finder method in the bean's local-home interface, but not necessarily an equivalent `ejbFind`*`Xxx`*`()` method in the bean itself (and certainly not in the code written by the bean provider).

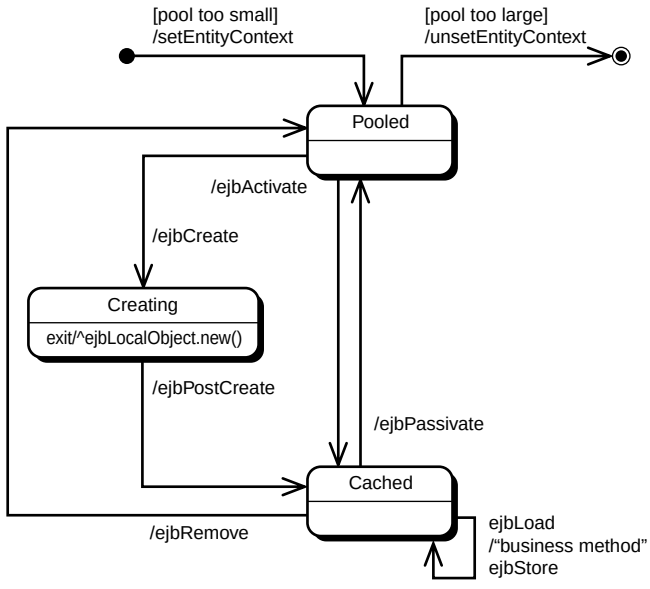


FIGURE 7.3

The `javax.ejb.EntityBean` lifecycle for CMP Entity beans.

Where the implementation of the finder methods is will depend on the EJB container. It does seem likely that many vendors will choose to place the finder logic in the bean's concrete implementation. An advantage of this approach (for the EJB container vendor) is that the rest of the EJB container need not differentiate between BMP and CMP Entity beans.

Yesterday, you saw the responsibilities of the bean provider for each of these lifecycle methods for the `Job` bean when implemented using BMP. The implementation is somewhat simpler when using CMP, as shown next.

As for BMP Entity beans, the `setEntityContext()` and `unsetEntityContext()` methods must be implemented to look up any required resources (although the resources are likely to be different than those needed for BMP; in particular, the JDBC DataSource should not be needed).

Under BMP, the `ejbCreate()` method was responsible for persisting the newly created bean's state to the persistent data store. Under CMP, the `ejbCreate()` method does not need to do this, but does still need to set the bean's fields to the parameters passed in. This can include generating a unique primary key value.

The BMP version of `ejbRemove()` was responsible for physically removing the bean's state from the data store. The CMP version does not need to do this (and later on, you'll see that this method actually has a null implementation).

Nevertheless, sometimes there may be occasions when work needs to be done in the `ejbRemove()` method. For example, you could imagine an Entity bean that has some subscribers interested in observing its changing state. One such Entity bean might be a headline news item publicized from Reuters. When this Entity bean is finally removed, it would notify those subscribers of the event.

On a more mundane level, the bean might want to prevent the delete from occurring if, for example, some referential integrity constraint would be violated by the bean being removed. In such cases, you need to know that the `ejbRemove()` is called *before* the container actually removes the data.

Next are the `ejbLoad()` and `ejbStore()` methods. Under BMP, these methods for the `Job` bean were substantial, because they had to read/write the bean's state to both the `Job` and `JobSkill` tables. Under CMP, unless there is any derived data to be maintained, these methods could well have an empty implementation.

The `ejbActivate()` and `ejbPassivate()` methods for a CMP Entity bean are pretty much identical to their implementations for a BMP Entity bean; after all, these methods have nothing to do with the persistent data store.

If you've been mentally (or manually) comparing this section with the previous section yesterday, you'll note that yesterday there was discussion on the finder methods. There is no such discussion here because the finder methods will be generated automatically. However, as the bean provider, you will need to indicate to the container the semantics of the finder queries. To do that, you must be familiar with EJB QL. But to understand EJB QL queries of any complexity, you need first to understand container-managed relationships. CMR is discussed in the next section, followed then by EJB QL.

Container-Managed Relationships

Container-managed relationships (CMR) might possibly sound pretty daunting, and certainly from the EJB container vendor's perspective, there could be some fairly complex activity happening behind the scenes. However, from the bean provider's perspective (that is, you), they are fairly straightforward and easy to use.

CMRs are defined declaratively through the deployment descriptor, underneath the `relationships` element. Therefore, container-managed relationships can only be defined between Entity beans that reside within the same local relationship scope (EJB specification,

section 10.3.2). What this means in practice is that beans that have relationships must be deployed in the same `ejb-jar` file. You will be learning more about actually declaring CMRs later today, in "Configuring a CMP Entity Bean."

The restriction that CMR can only be defined between EJBs deployed in the same `ejb-jar` file could possibly create problems. Some organizations maintain static reference data globally and replicate that data locally. This works because such data is often updated relatively infrequently.

By its nature, reference data is referenced (!), so one would expect relationships from domain-specific Entity beans up to cross-organizational reference beans. However, if the reference Entity beans are deployed separately from the domain-specific Entity beans (as would be likely), no such relationships can be assigned.

Relationship Types

CMR allows three different types of relationships to be defined between Entity beans:

- One-to-one
- One-to-many
- Many-to-many

The first two relationship types are to be expected, but the last is perhaps more unexpected if you are used to using RDBMS. In relational theory, it is not possible to create a many-to-many relationship directly; instead, a link (or association) table is required. Indeed, such a table can be seen directly in the BMP version of the case study database; the many-to-many link between jobs and skills is captured in the `JobSkill` table. However, the EJB specification allows many-to-many links to be defined directly for Entity beans, an immediate simplification over the RDBMS approach.

Of course, most EJB containers—the J2EE RI included—will persist to RDBMS, so will require a link table in the physical schema. Indeed, the J2EE RI uses a link table even for one-to-many associations. You can see this if you look back to Figure 7.2. For example, the one-to-many link from `Job` to `Location` is captured through the snappily named `JobBean_location_LocationBean_Table` table.

These relationship types actually refer to the maximum cardinality (also sometimes called multiplicity) of the related beans in the relationship. That is, saying that there is "a one-to-many relationship between `Location` and `Job`" is shorthand for "the maximum number of `Location`s that a `Job` can be related to is one, and the maximum number of `Job`s that a `Location` can be related to is many." There is also the question of minimum cardinality. In other words, is it necessary for a `Job` to be related to any `Location` (or can it be related to none)? Equally, must a `Location` have any `Job` related to it?

The EJB specification answers this question implicitly by always allowing a minimum cardinality of zero. Hence, "one-to-many" also allows for none-to-many, one-to-none, one-to-many, and (trivially) none-to-none.

There are sometimes situations when a minimum cardinality of none is not acceptable. For example, it might be the case that a `Job` must always relate to a `Location`. (Actually, for the case study, this is not enforced except in principle). In these cases, it is up to the bean to do the appropriate validation. In other words, the `Job` bean would only define a `create()` method that accepted a `Location` bean reference, and if it provided a `setLocation()` accessor method in its interface, it would ensure that the supplied `Location` reference was not `null`.

A related question is, "What happens if a bean is removed?" Suppose that the `Job` bean relates to a `Location`, and the `Location` bean is deleted. The `Job` bean will be left with a `null` reference. In relational terms, this is sometimes called a *cascade null*.

Suppose (again) that every `Job` must always relate to a non-`null` `Location`. There are a number of options:

- The first, somewhat radical, option is to remove the related `Job` beans—in other words, perform a cascade delete. CMR supports this directly (it is specified through the deployment descriptor) and will remove each `Job` bean in turn.
- Second, the application can prevent the removal of the foreign key `Location` bean from occurring. This would be done by implementing an appropriate check in the `ejbRemove()` lifecycle method.
- Another alternative would be to reassign every impacted `Job` bean to some new `Location`. Again, the `ejbRemove()` method would need to do this work.

The second option is probably the most likely, so you should take care to do this type of minimum cardinality analysis to make sure that you do not unwittingly end up with beans that have `null` relationships when the semantics of the problem domain prohibit this from occurring.

Navigability

In addition to specifying multiplicity of the relationships, CMR also allows the navigability of the relationship to be defined. The navigability is either unidirectional or bidirectional.

Navigability is defined by indicating the name of the field that references the related bean. For example, in a many-to-one relationship between the `Job` and `Location` beans, indicating a field of `location` for the `Job` bean means that there is navigability from `Job` to `Location`. There may not necessarily be navigability in the opposite direction; that would depend on whether the `Location` bean defines a field called `jobs`.

While the case study does not require bi-directional navigability either from `Location` to `Job` or from `Skill` to `Job`, it does define navigability nevertheless. Otherwise, the code generated by the J2EE RI 1.3 deployment tools (somewhat unfortunately) does not compile—not something to inspire confidence!

The term "field" (or more properly, `cmr-field`) used here indicates the accessor methods for the virtual fields defined in the CMP superclass and implemented by the EJB container. The next section looks at these methods in more detail.

cmr-fields

By way of example, the `location` `cmr-field` for the `Job` bean has accessor methods of `getLocation()` and `setLocation()`, and the `skills` `cmr-field` has the accessor methods `getSkills()` and `setSkills()`. The return type of these methods depend on the multiplicity of the relationship.

The relationship from `Job` to `Location` is many-to-one, so the methods that correspond to the `location` `cmr-field` are as follows:

```
public abstract LocationLocal getLocation();
public abstract void setLocation(LocationLocal location);
```

The same restriction on naming that applies to `cmp-fields` also applies to `cmr-fields`: the name must not start with a capital letter.

For single-valued `cmr-fields`, the return type for the getter and the parameter to the setter is the local interface of the related bean (`LocationLocal` in this case). Yesterday, it was noted that local interfaces are the cornerstone of container-managed relationships; this shows why. Remote interfaces cannot be used in CMR.

This isn't recommended, but there is nothing in principle to prevent an Entity bean with only a remote interface from having relationships with other Entity beans. However, the target Entity beans must themselves have a local interface, and the relationship will be unidirectional. The absence of a local interface in the source Entity bean prevents the related Entity beans navigating back to the Entity bean.

The `Job` bean also has a relationship with the `Skill` bean, this time many-to-many. Thus, the `skills` `cmr-field` corresponds to the following methods:

```
public abstract java.util.Collection getSkills();
public abstract void setSkills(java.util.Collection skills);
```

This is a multi-valued `cmr-field` because a collection of values is returned, not just a single value. The collection returned here is a `java.util.Collection` of references to the local interface of the related bean (`SkillLocal` in this case). The EJB specification also allows for `java.util.Sets` to be returned. The EJB specification does not currently allow `Lists` or `Maps` to be returned from multi-valued `cmr-fields`, but does hint that they may be added in the future.

Note that the fields of a bean are either regular `cmp-fields` or they are `cmr-fields` (or they are just regular instance variables, not managed by the container at all). Put another way, `cmp-fields` cannot be defined that have references to other beans as their argument or return type; such fields must be defined as `cmr-fields`.

Composite Primary Keys and Relationships

The `Job` Entity bean has a relationship with both the `Location` bean and the `Customer` bean. The `getCustomer()` method returns the name of a customer as a `String`, whereas the `getLocation()` method returns a reference to a `LocationLocal`. In other words, the former returns the name (the primary key) to a bean, and the latter returns the bean itself. So why the difference?

The reason is that the `customer` field is part of the primary key for the `Job` bean, and appears in `JobPK`. Every public field in `JobPK` must have a corresponding field in the bean itself.

If the `JobPK` class defined its `customer` field to be a reference to a `CustomerLocal`, the `JobPK` class could not be guaranteed to be serializable.

This shows up a very subtle area, not highlighted at all in the EJB specification. In the case study, there is a relationship between `Customer` and `Job`, in that `Job`s are identified by `Customer`. In other words, the primary key of `Job` contains the primary key of the `Customer` that "owns" that `Job`.

The case study does not define the one-to-many relationship between `Customer` and `Job`. If this had been done, a virtual field and corresponding accessor methods (`{get/set}CustomerObj()` methods) would need to have been defined. The problem that would then have arisen, however, is that potentially the name of the customer returned by `getCustomer()` may not correspond to the actual customer returned by `getCustomerObj()`.

In commercial EJB containers, this problem can be resolved by mapping both the `getCustomer()` and `getCustomerObj()` methods to the same physical data in the persistent data store (the `customer` column in the `Job` table). This prevents them from getting out of step (although even here, the EJB container would need to make the `customerObj` field read-only because allowing it to be changed would implicitly change the primary key of the `Job` bean).

However, the J2EE RI container does not make the mapping of the Entity beans data to the physical schema explicit. While it might be possible to modify the implied mapping, it would be unclear (from an education standpoint) what was being done. For this reason, the case study does not define the `Customer`/`Job` relationship. This is why, for example, the `customerObj` field is derived from the `customer` field, and is looked up in the `JobBean`'s `ejbLoad()` method.

Not only do the methods corresponding to the `cmr-field` return and accept only local and not remote interfaces to beans, they also cannot appear in the remote interface of the bean. This is not really surprising. After all, the return types and arguments to the methods corresponding to the `cmr-field` take only local interfaces of remote beans, so the client of the bean invoking the `cmr-field` methods must be local. You saw yesterday that there are very good reasons why remote interfaces are bad news for Entity beans; this is another reason not to provide a remote interface.

The other reason that `cmr-field` methods cannot appear in the remote interface is to prevent untoward network traffic. The performance cost of transporting large collections of references across the wire (even assuming those references were serializable) would be overwhelming.

Table 7.1 compares the use of `cmp-field`s and `cmr-field`s in interfaces.

TABLE 7.1 `cmp-field`s and `cmr-field`s and Interfaces

Feature	*cmp-field*	*cmr-field*
Can appear in local interface	Yes	Yes
Can appear in remote interface	Yes Not recommended though; Entity beans should be accessed via remote clients.	No
Can accept as parameters and return local references to beans	No `cmp-field`s deal only with primitives (or serializable objects).	Yes
Can accept as parameters and return remote references to beans	No `cmp-field`s deal only with primitives (or serializable objects).	No Container managed relationships are defined only through local interfaces of beans.

The EJB specification also requires that the `Collection` returned by a `cmr-field` method is only used in the same transaction context. You will be learning more about transactions tomorrow, but for now, just consider the following scenario. A Session bean could invoke a multi-valued `cmr-field`'s getter method and receive back a `Collection`. It could then hold onto this `Collection` for a few seconds, minutes, days, or months. It might also remove and add elements to this `Collection`. If the client then calls the setter method for the bean, there are no guarantees that either the `Collection` hasn't been changed by some other client of the entity bean (the so-called lost update problem) or that the elements in the originally returned collection are still valid (the repeatable read problem). The EJB specification's insistence that collections are only manipulated within a transaction solves these problems, primarily because transactions both prevent the items in the `Collection` from being removed, and ensure that new items will remain valid.

Manipulating Relationships

In the context of an RDBMS, relationships between tables can be manipulated in several ways. Consider, for example, the case study (with the schema used in Day 6, "Entity EJBs"). The `Job` and `Skill` tables are in a many-to-many relationship, resolved through the `JobSkill` link table, and the `Job` and `Location` table have a many-to-one relationship, implemented through the `location` column acting as a foreign key in the `Job` table.

To change a many-to-many relationship means adding or removing entries from the appropriate link table. In the example, this means adding or removing `(job, skill)` tuples from the `JobSkill` table.

To change a many-to-one relationship means changing the value of the foreign key column. In the example, this means changing the value of the `location` column in the `Job` table.

Manipulating relationships between Entity beans is somewhat different. As you have seen, the setter method for a multi-valued `cmr-field` (such as `setSkills()`) takes an entire `Collection` of items. And for many-to-one relationships (with the appropriate navigability), the relationship can be modified from the "parent" end, just as much as from the "child" end.

Moreover, for multi-valued `cmr-field`s, the collection of beans referenced by the relationship can be modified just by using the usual `add()` or `remove()` methods of the `java.util.Collection` interface.

The EJB specification lays out in some detail the semantics of various actions that impact relationships between Entity beans. Many of these are straightforward and act as expected, but a few deserve special comment.

Figure 7.4 shows an example configuration of `Location` and `Job` beans, and indicates the relationships before and after executing:

`loc1.getJobs().add(job2`$_1$`)`

The relationships indicated `{new}` are created as a result of this action, while the relationships indicated `{destroyed}` disappear. (This rather elegant notation is an enhancement to UML, described by deSouza and Wills' Catalysis Method.)

Initially, `loc1` is associated with `job1`$_1$ through `job1`$_n$, and similarly, `loc2` is associated with `job2`$_1$ through `job2`$_n$. You can see that as a result of the action, the `job2`$_1$ bean is added to the collection of jobs associated with the `loc1` location. However, perhaps less obviously, the `job2`$_1$ bean is also *removed* from the collection of jobs associated with the `loc2` location. This is because there is a one-to-many relationship between `Location` and `Job`, so the `job2`$_1$ bean cannot have a relationship with both `loc1` and `loc2` at the same time.

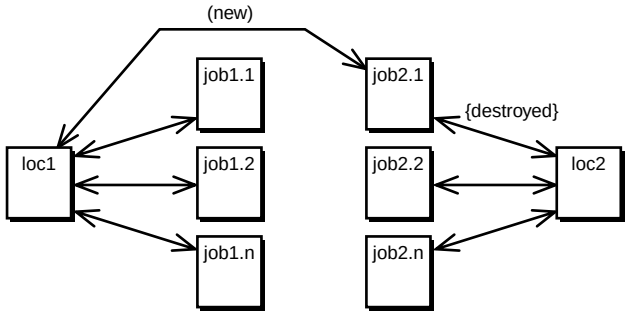


FIGURE 7.4

Before and after object diagram for `loc1.getJobs().add(job2`$_1$`)`.

Note that the collection of jobs associated with `loc1` changes, even though the `setJobs()` method of `Location` is not called! At least, these are the semantics laid out by the EJB specification, but it would be wise to check that your own EJB container correctly implements this. (The J2EE RI server does implement this correctly.)

The `java.util.Collection` interface defines the `addAll()` method as well as the `add()` method, and this is also supported for CMR collections. This takes a collection of elements rather than a single element. For one-to-many associations, the `addAll()` has the same semantics as `add()` in that any child elements (the `Job` bean in the example) that are added to some collection will also be removed from the collection where they resided.

The diagram in Figure 7.4 holds good for either unidirectional relationships with navigability from `Location` to `Job`, and for bi-directional relationships.

Figure 7.5 shows the "opposite" case of removing a bean from a collection, having executed

`loc1.getJobs().remove(job1`$_n$`)`

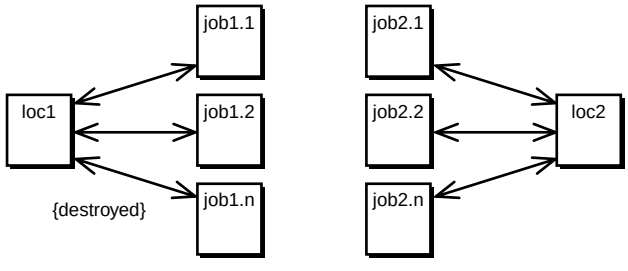


FIGURE 7.5

Before and after object diagram for `loc1.getJobs().remove(job1`$_n$`)`.

Initially, loc1 is associated with $job1_1$ through $job1_n$, and similarly, loc2 is associated with $job2_1$ through $job2_n$. As you can see, after executing this action, the $job1_n$ bean is no longer associated with any Location bean. Earlier today, it was noted that the EJB specification always allows a minimum multiplicity of zero, and this is what has occurred here. Suppose though that every Job should always be related to a Location—that is, a minimum multiplicity of one. Given that the job's location is being set to null indirectly (by calling remove() on the returned Collection from the getJobs() method), there is no easy way to enforce this business rule.

The only available solution is to not make the getJobs() method available in the interface of the bean. You could provide another method, perhaps named getJobsCopy(), that returns a copy of the Collection and make it clear that adding or removing beans to this Collection will not influence the relationship itself. Even better, the Collection could be made immutable. The following is a possible implementation for this method (assuming java.util.* is imported):

```
public Collection getJobsCopy() {
    List jobs = new ArrayList();
    for(Iterator iter = getJobs().iterator(); iter.hasNext(); ) {
        jobs.add(iter.next());
    }
    return Collections.unmodifiableList(jobs);
}
```

Incidentally, another way of removing elements from the collection returned by a getter method is to do so using an Iterator. Indeed, changing the contents of a Collection, either directly by using remove() or indirectly, such as by the use of add() as described in figure 7.4, will invalidate any Iterators instantiated from that Collection. In any case, the following would disassociate the loc1 location from all of its jobs:

```
for(Iterator iter = loc1.getJobs().iterator(); iter.hasNext(); ) {
    iter.next();
    iter.remove();
}
```

The diagram in Figure 7.5 again holds good for either unidirectional relationships with navigability from Location to Job and for bi-directional relationships.

The case study does not define any one-to-one relationships between beans, so imagine that there is an Employee bean that has a one-to-one relationship with Job. This could perhaps represent the Employee of the Advertiser who originally placed the Job advert.

In any case, Figure 7.6 shows the object diagram having executed

```
job1.setEmployee(job2.getEmployee())
```

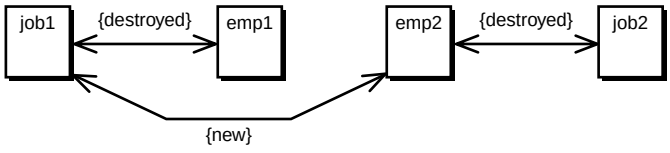


FIGURE 7.6
Before and after object diagram for `job1.setEmployee( job2.getEmployee() )`.

Initially, `job1` has a relationship with `emp1`, and `job2` has a relationship with `emp2`. After the action, `emp1` has no relationship with any `Job`, `job2` has no relationship with any `Employee`, and `job1` has a relationship with `emp2`.

This seems quite straightforward, but again, a minimum multiplicity of zero is needed. If either every `Employee` must always relate to a `Job`, or if every `Job` must always relate to an `Employee`, there is no method where this application validation can be performed. The `Job` bean's `setEmployee()` method might look like a good candidate, until you realize that this method is abstract and is generated entirely by the beans' subclass.

The solution, again, must be to not expose the setter method in the interface of the bean. Instead, a method, such as `setEmployeeField()`, could be provided. Its implementation, for the case where every `Job` must relate to an `Employee` (though not every `Employee` need have a relationship with a `Job`), might be as follows:

```
public void setEmployeeField(EmployeeLocal newEmployee) {
    if (newEmployee.getJob() != null) {
throw new IllegalArgumentException(
        ➥"New employee must not already be related to a job");
    }
    setEmployee( newEmployee );
}
```

As you can see, this implementation requires that the supplied employee is not related to a job. (If it were, that job would, in turn, need to be assigned a new employee not related to any job, and so on.)

Clearly, this technique is good wherever some application-level validation is required. For example, if a bean had a `cmp-field` called `phoneNumber`, the format of the supplied number could be checked.

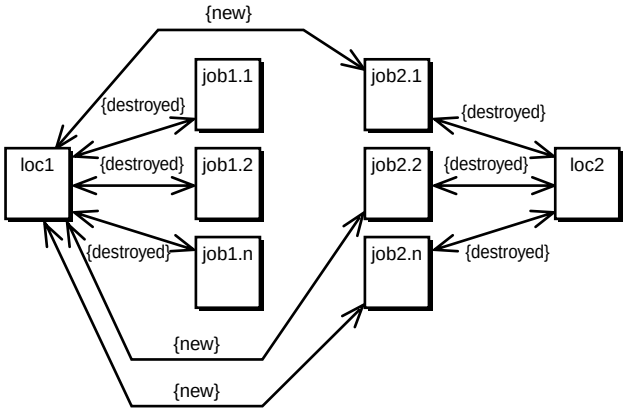
The diagram in Figure 7.6 holds good for unidirectional relationships with navigability from `Job` to `Employee` and for bi-directional relationships.

Figure 7.7 returns to the relationship between the `Location` and `Job` beans, this time showing the object diagrams having executed

```
loc1.setJobs(loc2.getJobs())
```

FIGURE 7.7

Before and after object diagram for `loc1.setJobs(loc2.getJobs())`.



Again, initially `loc1` is associated with $job1_1$ through $job1_n$, and similarly, `loc2` is associated with $job2_1$ through $job2_n$. After the action has completed, the `Collection` of jobs previously associated to `loc2` are associated to `loc1`. The `loc2` bean itself, and the `Collection` of jobs previously associated with `loc1`, no longer have any associations.

The discussion for the diagram in Figure 7.7 follows similar lines to that for the previous diagram in Figure 7.6. Specifically, handling minimum cardinality of one for either `Job` or `Location` will require the `setJobs()` method to be removed from the interface of the bean. Instead, a method, such as `setJobsField()`, should be exposed that will do the validation and then delegate to `setJobs()`.

It's also worth remarking that behind the scenes, the EJB container is having to perform some significant updates to the persistent data store. Assuming an RDBMS physical schema of `Location` and `Job` tables with a foreign key in the `Job` table, the foreign key for $job1_1$ through $job1_n$ would need to be set to `null`, and the foreign key for $job2_1$ through $job2_n$ updated to be `loc1`.

You saw earlier today that the actual physical schema used by J2EE RI actually has a separate link table called `JobBean_location_LocationBean_Table` that holds the foreign key. For the J2EE RI to implement this action, it would need to delete all the rows in this table for `loc1` and `loc2`, and then re-insert new rows corresponding to the `(loc1, job2`$_1$`)`... `(loc1, job2`$_n$`)` tuples.

If you were to design a bean with multi-valued relationships without using CMR, you might well have provided methods such as `addSkill()` and `removeSkill()`. As you now appreciate, CMR itself does not require or provide such methods, relying instead on a single setter method for the `cmr-field`.

Of course, there is nothing to prevent providing methods, such as `addSkill()` and `removeSkill()`, in the bean itself. Indeed, as you have seen, there are good reasons why the `setSkills()` method might not appear in the local interface to the bean; providing just `addSkill()` and `removeSkill()` in the interface would provide an (arguably) more natural interface to the bean while also allowing the bean to perform any application-specific validation.

Finally in this section, note that it is illegal to call the setter method of a multi-valued `cmr-field` with `null`; a `java.lang.IllegalArgumentException` will be thrown by the EJB container if this is attempted. In the example, to indicate that a job has no skills associated with it, an empty collection (for example, `java.util.Collection.EMPTY_LIST`) must be passed as the argument to `setSkills()`.

EJB QL

By now, you should have a good idea as to how a bean's CMP fields and CMR fields are managed and persisted by the EJB container. However, there is still one area that you covered yesterday when you were learning about BMP Entity beans that has not yet been touched on today, and that is how to specify finder methods. For this, you need to know something about the EJB Query Language, or EJB QL.

EJB QL was introduced in the EJB 2.0 specification to make beans more portable across EJB containers. Previously, each EJB container vendor created their own proprietary mechanism for specifying the semantics of finder methods. This is no longer the case.

As a language, EJB QL is based on ANSI SQL, which you have seen embedded within the case study code (for example, on Day 5, "Session EJBs" and Day 6, "Entity EJBs"). EJB QL also bears some similarity with the Object Constraint Language (OCL), part of UML. Familiarity with ANSI SQL (and ideally OCL too) will see you well on the way to picking up EJB QL.

Select Methods

The EJB 2.0 specification also introduces another concept that also uses EJB QL queries, namely that of select methods. These are like finder methods in that their purpose is to return data from the persistent data store, but there are also some important differences. Table 7.2 is an expanded version of the comparison that appears in the EJB specification.

TABLE 7.2 Finder and Select Methods Compared

Feature	*Finder Method*	*Select Method*
Can be defined for CMP Entity beans	Yes	Yes
Can be defined for BMP Entity beans	Yes	No
Semantics specified using EJB QL query, implementation generated by container	Yes	Yes
Appears in local-home (home) interfaces	Yes This is the way in which formal arguments are specified.	No
Abstract method signature in bean class	No	Yes This is the way in which the formal arguments are specified.
Returns local (remote) references to bean	Yes Local references are returned if specified on local-home interface, remote references if specified on home interface.	Yes By default, local references are returned, but remote references can be returned if indicated in the deployment descriptor (the `result-type-mapping` element).
Returns any `cmp-field` or `cmr-field`	No	Yes A different syntax of EJB QL query is used.
Returns `java.util.Collections` of references	Yes	Yes
Returns `java.util.Sets` of references	No Duplicates can be eliminated using the `distinct` keyword in the EJB QL query.	Yes
Returns `java.util.Enumerations` of references	Yes This is to support legacy EJB 1.x beans; do not use.	No
Called on pooled beans	Yes The EJB container selects an unused bean to perform the query.	Yes If being called from a home method.

TABLE 7.2 Continued

Feature	*Finder Method*	*Select Method*
Called on activated beans	No	Yes If being called from a business method, this allows the underlying query to be logically scoped (for example, based on the primary key), if needed.
Own transaction context can be specified	Yes	No Because the select method does not appear in the interfaces, no transaction context can be defined. Instead, the context of the calling method is used.

You'll be learning how to write select methods later, as part of specifying and implementing CMP beans.

Syntax and Examples

The EJB Specification formally defines the syntax of EJB QL queries using Bacchus Normal Form (BNF). This can be somewhat heavy going to wade through, but is comprehensive. This section introduces EJB QL using BNF, with annotations and examples along the way.

An EJB QL query is defined as follows:

```
EJB QL ::= select_clause from_clause [where_clause]
```

This says that a query consists of a `select_clause` and a `from_clause`, and optionally a `where_clause`. Immediately, you can see the similarity with ANSI SQL.

In ANSI SQL, you may recall that the columns listed in the `SELECT` clause relate to the tables identified in the `FROM` clause. So in exploring EJB QL, it makes sense to look at the `from_clause` before the `select_clause`; after all, both are needed for any well-formed EJB QL query.

from_clause

The `from_clause` is defined as follows:

```
from_clause ::= FROM identification_variable_declaration
                ➥[, identification_variable_declaration]*
```

This says that in an EJB QL query, the `from_clause` consists of one or more `identification_variable_declaration` clauses. This corresponds to ANSI SQL where there are one or more tables/views listed in a SQL `FROM` clause. The `identification_variable_declaration` clause is then defined as follows:

```
identification_variable_declaration ::=
        ➥{range_variable_declaration | collection_member_declaration}
        ➥[AS ] identification_variable
```

where

```
range_variable_declaration ::= abstract_schema_name
```

and

```
collection_member_declaration ::= IN (collection_valued_path_expression)
collection_valued_path_expression ::=
        ➥identification_variable.
        ➥[single_valued_cmr_field.]*
        ➥collection_valued_cmr_field
```

In other words, each item in the `from_clause` either is just an `abstract_schema_name`, or is an expression of the form `IN (o.abc.def.xyz)`. The `abstract_schema_name` is probably the easier expression to understand. Each CMP Entity bean has a corresponding `abstract_schema_name`, ultimately declared in the deployment descriptor. So in ANSI SQL terms, this is very similar to just listing the "table" that corresponds to the bean.

The `IN (o.abc.def.xyz)` expression is similar to an OCL expression. Here `o` refers is the `identication_variable` assigned by the previous `AS` phrase. In other words, it corresponds to some `abstract_schema_name` (that is, bean) also in the `from_clause`. The `abc` and `def` are `single_value_cmr_fields`. In other words, these are fields of the `o` bean that return a single element. The `abc` is a field of `o`, returning a reference to a bean, and `def` is a field of this reference that has a field `xyz`. This `xyz` field, in turn, returns a `Collection` or `List` of some other data (a bean or otherwise).

That's pretty heavy, so some examples from the case study may clarify things. Assuming that the `Job` bean has an abstract schema name called `Job`, that the `Customer` bean has an abstract schema name called `Customer`, and so on.

```
FROM Job AS j
```

sets up an `identification_variable` called `j` that refers to the `Job` schema. The comparison with ANSI SQL is obvious; the syntax is the same.

```
FROM Job AS j, IN (j.skills) AS s
```

sets up the `j identification_variable` as before and also an `identification_variable` called `s` that refers to each of the `skills` related to the `Job` bean in turn.

Comparing this to ANSI SQL, this most likely would correspond to

```
FROM       Job AS j
INNER JOIN JobSkill AS s
        ON s.customer = j.customer
       AND s.ref = j.ref
```

or if you prefer the old-fashioned way:

```
FROM Job AS j, JobSkill AS s
WHERE s.customer = j.customer
  AND s.ref = j.ref
```

The following is another example:

```
FROM Job AS j, IN (j.location.jobs) AS k
```

sets up the `j identification_variable` as before and also an `identification_variable` called `k` that refers to all of the jobs that have the same location as the original job. To see this, note that `j.location` returns a reference to the `Location` bean for the `Job`, and then that `j.location.jobs` returns the `Collection` of `Jobs` for that `Location`. Of course, this `Collection` will include the original job, but it will include others as well.

Comparing this to ANSI SQL, you can see that this is a self-join:

```
FROM Job AS j
INNER JOIN Job AS k
        ON j.location = k.location
```

or in the old money:

```
FROM Job AS j, Job AS k
WHERE j.location = k.location
```

select_clause

The `select_clause` is defined as follows:

```
select_clause ::= SELECT [DISTINCT ] {single_valued_path_expression |
                  ➥OBJECT (identification_variable)}
```

The easy case is the `SELECT OBJECT(o)` style of the `select_clause`, where `o` is an `identication_variable` defined by the `from_clause`. (You see now why the `from_clause` was presented first.) This returns all the data from the data store required to instantiate a bean. In ANSI SQL terms, you might think of it as an intelligent `SELECT * FROM ...`.

All finder methods must use the `SELECT OBJECT(o)` style, where the objects returned are of the schema associated with the bean for which the finder is being specified. The other style of the `select_clause`, using a `single_valued_path_expression` is for use only by select methods that you were introduced to briefly earlier. (More on these to follow.)

The following are some examples from the case study.

```
SELECT OBJECT(j)
FROM Job as j
```

will return all jobs. If the `JobLocalHome` interface defined a finder method called `findAll()`, this would be the corresponding EJB QL query.

The next example

```
SELECT DISTINCT OBJECT(s)
FROM Job as j, IN (j.skills) as s
```

will return back all skills used by any job. Because some skills will be required by more than one job, the `DISTINCT` keyword is used to eliminate duplicates. This EJB QL query might perhaps be associated with a finder method called `findAllRequiredSkills()` on the `SkillLocalHome` interface.

The other style of the `select_clause` uses a `single_valued_path_expression`. This is defined as follows:

```
single_valued_path_expression ::=
        ➥{single_valued_navigation | identification_variable}
        ➥.cmp_field | single_valued_navigation
```

where a `single_valued_navigation` is

```
single_valued_navigation ::= identification_variable.[single_valued_cmr_field.]*
➥single_valued_cmr_field
```

Taking these definitions together, a `single_valued_path_expression` is effectively just a chain of (none or many) `single_valued_cmr_fields` (`cmr-field`s returning a reference to a single bean, and not a collection), eventually finishing with a `cmp-field`.

In the following examples

```
SELECT DISTINCT j.location.name
FROM Job as j
```

`location` is the `cmr-field` of the `Job` schema (identified by `j`), and `name` is the `cmp-field` of the bean referenced by the `cmr-field` (a `Location` bean, obviously).

This returns the names of the locations where there are jobs. Comparing this to ANSI SQL, you can see that EJB QL is actually simpler (because of its use of the OCL-like path expressions to navigate between beans):

```
SELECT DISTINCT l.name
FROM Job as j INNER JOIN Location as l ON j.location = l.location
```

However, the following is not allowed:

```
SELECT DISTINCT j.skills.name
FROM Job as j
```

This is because the `skills cmr-field` of `Job` returns a collection of skills, not a single skill. The correct way to phrase this query is as follows:

```
SELECT DISTINCT s.name
FROM Job as j, IN (j.skills) AS s
```

You might like to think of the `s` identification variable as an iterator over the collection of skills returned by the `skills cmr-field`.

Note that the `select_clause` in EJB QL can only ever return a single item of information, so the following also is not allowed:

```
SELECT DISTINCT j.location.name, j.location.description
FROM Job as j
```

where_clause

The `where_clause` is optional in an EJB QL query, but will be present in the majority of cases. It is defined in BNF as follows:

```
where_clause ::= WHERE conditional_expression

conditional_expression ::=
        ➥conditional_term | conditional_expression OR conditional_term
conditional_term ::=
        ➥conditional_factor | conditional_term AND conditional_factor
conditional_factor ::=
        ➥[NOT ] conditional_test
```

This just says that clauses can be combined using the usual `AND`, `OR`, and `NOT`. Much of the rest of the formal BNF definitions for the `where_clause` also makes somewhat heavy work of some fairly straightforward concepts, so to paraphrase,

- `conditional_tests` can involve `=`, `>`, `<`, `>=`, `<=`, and `<>` operators. These apply variously to numbers and datetimes (all operators), and strings, Booleans, and Entity beans (the `=` and `<>` operators). You will recall that Entity beans are considered identical if their primary keys are equal.
- Comparisons can involve input parameters, where these correspond to the arguments of the finder or select method. More on this topic shortly.
- Arithmetic expressions can use the `BETWEEN...AND` operator, just as in ANSI SQL.
- String expressions can be compared against lists using the `IN` operator and against patterns using the `LIKE` operator. These also exist within ANSI SQL. Unlike Java, string literals should appear in *single* quotes.
- The `IS NOT NULL` operator exists to determine if an object is `null`. Again, this syntax is borrowed from ANSI SQL.

ANSI SQL supports the concept of nullable primitives (`int`s and so on), whereas Java and EJB QL do not. However, nullable primitives can be simulated by using wrapper classes as `cmp-field`s and using these in EJB QL expressions.

There are some more operators to EJB QL and some built-in functions, but first, some examples using these operators are in order.

```
SELECT OBJECT(c)
FROM Customer AS c
WHERE c.name LIKE "J%"
```

This will find all customers whose name begins with the letter J. Note that the ANSI SQL wildcards (`%` to match none or many characters, _ to match precisely one character) are used.

The following

```
SELECT OBJECT(j)
FROM Jobs AS j
WHERE j.location IS NULL
```

will find all jobs where the location has not been specified.

```
SELECT l.description
FROM Location AS l
WHERE l.name IN ("London", "Washington")
```

returns the descriptions of the locations named `London` and `Washington`.

The `where_clause` can also include input parameters. These parameters correspond to the arguments of the finder or the select method, as defined in the local-home interface or bean, respectively.

For example, the `Job` bean declares the following finder method in the `JobLocalHome` interface:

```
Collection findByCustomer(String customer);
```

The EJB QL query for this finder method is as follows:

```
SELECT OBJECT(j)
FROM Job AS j
WHERE j.customer = ?1
```

`?1` acts as a placeholder, with the `1` indicating that the first argument of the finder method (the `customer` string) be implicitly bound to this input parameter. Unlike JDBC SQL strings, the number is required because the binding is implicit, not explicit.

It is also needed in the cases where a single argument is used more than once in the query. For example, consider the following finder method:

```
Collection findLocationsNamedOrNamedShorterThan(String name);
```

This might have an EJB QL query of

```
SELECT OBJECT(l)
FROM Location AS l
WHERE l.name = ?1
OR LENGTH(l.name) < LENGTH(?1)
```

This example uses the built-in function `LENGTH` that returns the length of a `String`. In any case, this rather peculiar finder method will find those locations that have the exact name, and will also return any name whose length is strictly shorter than the supplied `name`. You can see that the `?1` placeholder appears more than once because the name argument needs to be bound to the query in two places.

EJB QL defines just a few more built-in functions. The functions that return a string are `CONCAT` and `SUBSTRING`. The functions that return a number are `LENGTH`, `ABS`, `SQRT`, and `LOCATE`. This last is effectively the same as `String.indexOf(String str, int fromIndex)`.

EJB QL defines two final operators—`IS [NOT] EMPTY` and `[NOT] MEMBER OF`. Neither of these have any direct equivalents in ANSI SQL, but both do have equivalents in OCL.

The `IS [NOT] EMPTY` operator can is similar to the `isEmpty` operator of OCL. It can be used to determine whether a collection returned by a `cmr-field` is empty. For example,

```
SELECT OBJECT(s)
FROM Skill AS s
WHERE s.jobs IS EMPTY
```

will return all those skills that are not marked as required by any job. This might be the query for a finder method on the `SkillLocalHome` interface, called something like `findNotNeededSkills()`.

In fact, this type of query can be expressed in ANSI SQL, though it does require a sub-query:

```
SELECT s.*
FROM Skill AS s
WHERE NOT EXISTS
    (   SELECT *
        FROM JobSkill AS j
        WHERE s.skill = j.skill )
```

The `[NOT] MEMBER OF` operator is similar to the `include` operator of OCL. Consider the following finder method on the `JobLocalHome` interface:

```
Collection findJobsRequiringSkill(SkillLocal skill);
```

The EJB QL query for this would be

```
SELECT OBJECT(j)
FROM Job AS j
WHERE ?1 MEMBER OF j.skills
```

Again, this can be expressed in ANSI SQL, but only using a subquery:

```
SELECT j.*
FROM Job AS j
WHERE EXISTS
    (   SELECT *
        FROM JobSkill AS s
        WHERE j.customer = s.customer
        AND   j.ref      = s.ref
        AND   s.skill    = ?1         )
```

Further Notes

You may have heard of the "OO/relational impedance mismatch." This is that to deal with objects, each must be instantiated and then a message sent to it. On the other hand, relational theory deals with sets of elements sharing some common attribute; to identify these elements without instantiating them effectively breaks encapsulation.

EJB QL does a pretty reasonable job of reconciling these concerns. By allowing queries to be expressed in a set-oriented syntax, the EJB container can easily map these to ANSI SQL when the persistent data store is an RDBMS. On the other hand, the generated implementations of finder methods return only objects or `Collections` of objects.

Nevertheless, EJB QL does have some limitations. For example, when constructing an EJB QL query, only declared relationships between beans can be followed. It is not possible to join arbitrary fields together (as it is in ANSI SQL). For example, those `Customers` who are also `Applicants` could not be identified using a condition such as `Applicant.name = Customer.name`.

There are a number of other cases where EJB QL is not (yet) as powerful as ANSI SQL. For example, EJB QL does not support grouping and aggregating, ordering, subqueries, and unions. Expect these features to be added as EJB QL matures. Also, even though EJB QL does not directly support subqueries, one might not be needed anyway thanks to the `IS [NOT] EMPTY` and `[NOT] MEMBER OF` operators.

Specifying a CMP Entity Bean

Specifying a CMP Entity bean is identical to specifying a BMP Entity bean; it consists of defining the local-home interface and the local interface. This makes sense; after all, to the user of the bean, it should not matter whether the bean is implemented internally using CMP or BMP.

The Local-Home Interface

For completeness, Listing 7.2 is the local-home interface of the `Job` bean.

LISTING 7.2 `JobLocalHome` Interface

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import java.util.*;
 5: import javax.ejb.*;
 6:
 7: public interface JobLocalHome extends EJBLocalHome
 8: {
 9:     JobLocal create (String ref, String customer) throws CreateException;
10:     JobLocal findByPrimaryKey(JobPK key) throws FinderException;
11:     Collection findByCustomer(String customer) throws FinderException;
12:     Collection findByLocation(String location) throws FinderException;
13:     void deleteByCustomer(String customer);
14: }
```

The Local Interface

Listing 7.3 shows the local interface for the `Job` bean; it, too, is unchanged from the BMP version.

LISTING 7.3 `JobLocal` Interface

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import javax.ejb.*;
 5: import java.util.*;
 6:
 7: public interface JobLocal extends EJBLocalObject
 8: {
 9:     String getRef();
10:     String getCustomer();
11:     CustomerLocal getCustomerObj(); // derived
12:
13:     void setDescription(String description);
```

LISTING 7.3 Continued

```
14:     String getDescription();
15:
16:     void setLocation(LocationLocal location);
17:     LocationLocal getLocation();
18:
19:     Collection getSkills();
20:     void setSkills(Collection skills);
21: }
```

Implementing a CMP Entity Bean

Just as for BMP Entity beans, implementing a CMP Entity bean involves providing an implementation for the methods of the `javax.ejb.EntityBean`, corresponding methods for each method in the home interface, and a method for each method in the remote interface.

Implementing `javax.ejb.EntityBean`

Under BMP, the `setEntityContext()` method was used to look up various bean home interfaces from JNDI, and the JDBC `DataSource` called `java:comp/env/jdbc/Agency` was also obtained. Because most of these relationships are now managed by the container, only a couple of home interfaces now need to be obtained, and there is no requirement to look up the `DataSource`. Listing 7.4 shows this.

LISTING 7.4 The `JobBean`'s `setEntityContext()` and `unsetEntityContext()` Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: import javax.naming.*;
 5: // imports omitted
 6:
 7: public abstract class JobBean implements EntityBean {
 8:
 9:     private EntityContext ctx;
10:
11:     public void setEntityContext(EntityContext ctx) {
12:         this.ctx = ctx;
13:         InitialContext ic = null;
14:         try {
15:             ic = new InitialContext();
16:             customerHome = (CustomerLocalHome)
                ➥ic.lookup("java:comp/env/ejb/CustomerLocal");
17:             jobHome = (JobLocalHome)
                ➥ic.lookup("java:comp/env/ejb/JobLocal");
```

LISTING 7.4 Continued

```
        }
        catch (NamingException ex) {
            error("Error looking up depended EJB or resource", ex);
            return;
        }
    }

    public void unsetEntityContext() {
        this.ctx = null;
        customerHome = null;
        jobHome = null;
    }

    // code omitted
}
```

The lookup of the `CustomerHome` home interface is required because the relationship from `Customer` to `Job` is maintained by the bean.

The lookup of the `JobHome` home interface is required only because the `ctx.getLocalHome()` method returns `NULL` for the J2EE RI 1.3. This would appear to be a bug. The `ejbHomeDeleteByCustomer()` home method actually uses the bean's own home interface.

As noted previously, under CMP, the `ejbLoad()` and `ejbStore()` methods often have empty implementations. If there is derived data to be maintained, it should be managed here. Indeed, the `JobBean` class does need to do this, as shown in Listing 7.5.

LISTING 7.5 The `JobBean`'s `ejbLoad()` and `ejbStore()` Methods

```
package data;

import javax.ejb.*;
import javax.naming.*;
// imports omitted

public abstract class JobBean implements EntityBean {

    public void ejbLoad() {
        JobPK key = (JobPK)ctx.getPrimaryKey();

        try {
            this.customerObj =
            ➥customerHome.findByPrimaryKey(getCustomer());
        }
        catch (FinderException e) {
            error("Error in ejbLoad (invalid customer) for " + key, e);
```

LISTING 7.5 Continued

```
17:         }
18:     }
19:
20:     public void ejbStore() { }
21:
22:    // code omitted
23: }
```

The `ejbLoad()` method is called after the bean's state has been populated, so the bean's state can be read through the accessor methods, if needed.

The `findByPrimaryKey()` method call on line 13 populates the `customerObj` field, using the value of `getCustomer()` accessor method. It's worth appreciating that `getCustomer()` returns just a `String`. In other words, this is the name (actually, the primary key) of a customer. To save business methods having to continually look up the actual customer that corresponds to this customer name, the `ejbLoad()` method does it once.

You can see that the `ejbStore()` method is trivial—there is nothing to do.

The `ejbActivate()` and `ejbPassivate()` methods have nothing to do with data stores, so their implementation is unchanged from the BMP version. This is shown in Listing 7.6.

LISTING 7.6 The `JobBean`'s `ejbActivate()` and `ejbPassivate()` Methods

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: // imports omitted
 5:
 6: public abstract class JobBean implements EntityBean {
 7:
 8:     public void ejbPassivate() {
 9:         setRef(null);
10:         setCustomer(null);
11:         customerObj = null;
```

LISTING 7.6 Continued

```
12:         setDescription(null);
13:         setLocation(null);
14:     }
15:
16:     public void ejbActivate() { }
17:
18:    // code omitted
19: }
```

Implementing the Local-Home Interface Methods

The methods of the local-home interface are implemented partly in code and partly through the provision of an appropriate EJB QL query. This section shows the queries that correspond to the finder methods in the home interface; the next section (configuring a CMP Entity bean) shows how to take those EJB QL strings and put them into the correct part of the deployment descriptor.

Create and Remove Methods

Listing 7.7 shows the `ejbCreate()` method for `Job` bean under CMP.

LISTING 7.7 The `JobBean`'s `ejbCreate()` Method

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: // imports omitted
 5:
 6: public abstract class JobBean implements EntityBean {
 7:
 8:     public JobPK ejbCreate(String ref, String customer)
                ➥throws CreateException {
 9:         // validate customer login is valid.
10:         try {
11:             customerObj = customerHome.findByPrimaryKey(customer);
12:         } catch (FinderException ex) {
13:             error("Invalid customer.", ex);
14:         }
15:
16:         JobPK key = new JobPK(ref,customerObj.getLogin());
17:         // for BMP, there was a workaround here,
               ➥namely to call ejbFindByPrimaryKey
18:         // under CMP, cannot call since doesn't exist.
19:         // instead, use jobHome interface ...
20:         try {
```

LISTING 7.7 Continued

```
21:             jobHome.findByPrimaryKey(key);
22:             throw new CreateException("Duplicate job name: "+key);
23:         }
24:         catch (FinderException ex) {}
25:
26:         setRef(ref);
27:         setCustomer(customer);
28:         setDescription(null);
29:         setLocation(null);
30:         return null;
31:     }
32:
33:     // code omitted
34: }
```

Note the use of accessor methods (the "setter" methods) to save the bean's state. This contrasts with the BMP equivalent where the fields were written to directly (for example, `this.ref = ref`).

The implementation of this method would have been even shorter if the `findByPrimaryKey()` call checking for duplicates had been omitted. Indeed, if an RDBMS is being used as the persistent data store (as it is for the case study), there is likely to be a unique index on the primary key on the appropriate table, meaning that any attempt to `INSERT` a duplicate would be detected.

Strictly, the appropriate exception to throw here is a `DuplicateKeyException`, not a `CreateException`. However, the EJB specification does not mandate this. Moreover, the specification even allows that the EJB container can defer any interaction with the persistent data store until the end of the transaction, and is not clear in this circumstance what type of exception should be raised.

As the EJB specification continues to mature, it will more fully define expected exceptions in different situations. Until then, beware that—regardless of the hype—moving from one EJB container to another may well throw up differences that will necessitate changes in your application.

The `ejbCreate()` method is called first, and then the container's concrete implementation persists the bean's state, and then the bean's `ejbPostCreate()` method is called.

Under CMP, the bean should return `null` from the `ejbCreate()` method. This compares to returning the actual primary key under BMP. Put another way, it doesn't matter what your method returns, it will be ignored. The reason for this is that the EJB container can access the information that constitutes the primary key anyway, by virtue of the `cmp-fields`.

The technical reason that the EJB container requires that CMP Entity beans return `null` is so that EJB container vendors can implement CMP by effectively subclassing the CMP bean and creating a BMP Entity bean (so far as the rest of the EJB container is concerned). Indeed, if you look at the generated code, this is precisely what the J2EE RI container does:

```
package data;

public final class JobBean_PM extends JobBean implements
com.sun.ejb.PersistentInstance {

public data.JobPK ejbCreate(java.lang.String param0,
java.lang.String param1) throws javax.ejb.CreateException {
com.sun.ejb.Partition partition =
com.sun.ejb.PersistenceUtils.getPartition(this);
         partition.beforeEjbCreate(this);
         super.ejbCreate(param0, param1);
         return (data.JobPK) partition.afterEjbCreate(this);
    }

    // code omitted

}
```

You can see the call to `super.ejbCreate()`. The return type is ignored, but the subclass' `ejbCreate()` does return a primary key to the rest of the EJB container.

The `ejbRemove()`method for the `Job` bean is shown in Listing 7.8.

LISTING 7.8 The `JobBean's ejbRemove()` Methods

```
1: package data;
2:
3: import javax.ejb.*;
4: // imports omitted
5:
6: public abstract class JobBean implements EntityBean {
7:
8:     public void ejbRemove() { }
```

LISTING 7.8 Continued

```
 9:
10:     // code omitted
11: }
```

As you can see, the implementation of `ejbRemove()` is trivial—there is nothing to do. Nevertheless, an implementation is required.

The BMP version of `ejbRemove()` for `Job` bean reset all the fields to `null`. Strictly speaking, there was no direct requirement for doing this, because when the bean instance is next used from the pool, its `ejbLoad()` will (should) populate all fields.

When implementing CMP Entity beans, you absolutely must not reset the fields to `null`. Doing so will cause the EJB container to throw an exception, because the bean's state is required so that the container can remove the correct data from the persistent data store.

Finder Methods

The implementation of the finder methods is by formulating appropriate EJB QL queries. The `JobLocalHome` interface defines three finder methods—`findByPrimaryKey()`, `findByCustomer()`, and `findByLocation()`.

The first bit of good news is that there is no need to define an EJB QL query for the `findByPrimaryKey()` method at all. You will recall that the `primkey-class` element is used in the deployment descriptor to indicate to the EJB container the class (either custom or pre-existing) that represents the primary key of the Entity bean. When there is a custom primary key class, the EJB container can use the structure of that class to figure out how to implement this method.

If there is no custom primary key class, an additional piece of information is required in the deployment descriptor—namely, the `primkey-field` element. This nominates the (single) `cmp-field` that represents the primary key for the bean.

If you are using a custom primary key class (such as `JobPK`), you do need to ensure that its public fields correspond exactly in name and type to a subset of the `cmp-field`s of the bean. The `JobPK` class is shown in Listing 7.9.

LISTING 7.9 JobPK Class

```
 1: package data;
 2:
 3: import java.io.*;
 4: import javax.ejb.*;
 5:
 6: public class JobPK implements Serializable
 7: {
 8:     public String ref;
 9:     public String customer;
10:
11:     public JobPK() {
12:     }
13:     public JobPK(String ref, String customer) {
14:         this.ref = ref;
15:         this.customer = customer;
16:     }
17:
18:     public String getRef() {
19:         return ref;
20:     }
21:     public String getCustomer() {
22:         return customer;
23:     }
24:
25:     // code omitted
26: }
```

The EJB container will match the `ref` and `customer` fields with `getRef()`/`setRef()` and `getCustomer()`/`setCustomer()` accessor methods for the `cmp-fields`. You can see here that the fields in the primary key class must be declared to have `public` visibility.

Moving onto the other finder methods, the findByCustomer() method has the following signature in the local-home interface:

```
Collection findByCustomer(String customer) throws FinderException;
```

The EJB QL query for this is as follows:

```
SELECT OBJECT(j)
FROM Job AS j
WHERE j.customer = ?1
```

The other finder method is findByLocation(), whose signature is as follows:

```
Collection findByLocation(String location) throws FinderException;
```

The EJB QL query for this is as follows:

```
SELECT OBJECT(;)
FROM Job AS ;
WHERE ;.location.name = ?1
```

Home Methods

The last method in the local-home interface is the home method `deleteByCustomer()`. This is used by clients when removing a customer; all of its jobs must also be removed. You've already seen the implementation of this home method under BMP, using SQL to delete from the `Job` and `JobSkill` tables. Under CMP, the implementation is somewhat more object-oriented, as shown in Listing 7.10.

LISTING 7.10 `JobBean`'s `ejbHomeDeleteByCustomer()` Method

```
 1: package data;
 2:
 3: import java.rmi.*;
 4: import java.util.*;
 5: import javax.ejb.*;
 6: // imports omitted
 7:
 8: public abstract class JobBean implements EntityBean {
 9:
10:     public void ejbHomeDeleteByCustomer(String customer) {
11:         try {
12:             Collection col = this.jobHome.findByCustomer(customer);
13:             for (Iterator iter = col.iterator(); iter.hasNext(); ) {
14:                 JobLocal job = (JobLocal)iter.next();
15:                 // remove job from collection
16:                 iter.remove();
17:                 // remove job itself
18:                 job.remove();
19:             }
20:         }
21:         catch (FinderException e) {
22:             error("Error removing all jobs for " + customer, e);
23:         }
24:         // needed because of the explicit job.remove()
25:         catch (RemoveException e) {
26:             error("Error explicitly removing job for " + customer, e);
27:         }
28:
29:     }
30:     // code omitted
31:
32: }
```

The method calls the bean's own `findByCustomer()` method, which returns a collection of jobs for the customer. It then iterates over these jobs and removes them one-by-one.

The `jobHome` field holds a reference to the bean's own home interface. In principle, the `entityContext.getEJBHome()` method could have been called, but the J2EE RI 1.3 container seems always to return `null`.

If there had been no suitable finder method, this would have been a prime case for using a select method. The select method would be declared in the `JobBean` class as follows:

```
package data;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
// imports omitted

public abstract class JobBean implements EntityBean {

    public abstract Collection ejbSelectByCustomer(String customer);

    // code omitted
}
```

The EJB QL query would be identical to that of the finder method. The only other difference would be in how the method was called. Rather than invoking the finder method through the home interface, the ejbSelect method could be called directly. So, in the ejbHomeDeleteByCustomer() method, the line

```
Collection col = this.jobHome.findByCustomer(customer);
```

would be replaced with

```
Collection col = this.ejbSelectByCustomer(customer);
```

In fact, the BMP and CMP implementations are not quite comparable; the CMP implementation is a little more robust. As was mentioned earlier, the BMP version of this method just deleted the appropriate rows from the `Job` and `JobSkill` table (turn back to yesterday's chapter to see this, if needed). This means that the `Job` bean's `ejbRemove()` method is never called; there is no opportunity for the bean to perform clean-up processing.

An alternative design again might have been to define a cascade delete relationship from `Customer` through to `Job`. The EJB Specification does require that `ejbRemove()` is called on every bean being deleted as a result of the cascade, so the net result is similar to the CMP implementation.

However, as has been remarked earlier, setting up a (cascade delete) relationship for `Customer` and `Job` is not easy when using the J2EE RI, because it does not easily support relationships between beans when one bean is identified by another (part of the primary key is also a foreign key).

A cascade delete relationship might be preferable to the CMP implementation. A naïve EJB container implementation would work the same way as the hand-coded CMP implementation, explicitly deleting each and every child bean one-by-one. The performance hit could be substantial.

A more sophisticated EJB container implementation ought to be able to call `ejbRemove()` for each bean, but then delete all of the child beans using a single call to the persistent data store; in other words, combining the best of the BMP and CMP approaches.

A related issue is that if a client happens to have a reference to a `Job` for the `Customer` being deleted, they won't find out that the `Job` has been removed until they attempt to access that `Job` again. At that point, the client will receive a `java.rmi.NoSuchObjectException`. But note that this isn't a problem just with BMP; the same behavior will occur for CMP also.

Implementing the Local Interface Methods

Looking back at the `JobLocal` interface back in Listing 7.3, you can see that many of the methods in the local interface simply expose the bean's `cmp-field`s or `cmr-field`s to its clients. Of course, there is no implementation for these methods because they are implemented by the EJB container's deployment tools. Therefore, all that is required is to copy the methods over from the local interface and mark them abstract. If you cast your eyes back all the way to Listing 7.1, you'll see that this is precisely what was done.

The only method in the `JobLocal` interface that does not correspond to an accessor method for a `cmp-field` or `cmr-field` is the `getCustomerObj()` method. Its implementation is shown in Listing 7.11.

LISTING 7.11 JobBean's `getCustomerObj()` Method

```
 1: package data;
 2:
 3: import javax.ejb.*;
 4: // imports omitted
 5:
 6: public abstract class JobBean implements EntityBean {
 7:     private CustomerLocal customerObj; // derived
 8:     public CustomerLocal getCustomerObj() {
 9:         return customerObj;
10:     }
11:     // code omitted
12: }
```

Pretty straightforward, though of course the hard work is done in `ejbLoad()` (Listing 7.5) and `ejbCreate()` (Listing 7.7). Recall that the `customerObj` field holds the actual reference to the "parent" `Customer` for the `Job` and is derived from the `customer cmp-field` that holds merely the name of the `Customer`. Because the customer `cmp-field` makes up part of the primary key, it is immutable, and so is the `customerObj` field—hence, no `setCustomerObj()` method.

Configuring a CMP Entity Bean

As you now appreciate, a substantial part of the implementation effort for CMP Entity beans is simply completing the deployment descriptor correctly.

This will always involve completing the `entity` element of the deployment descriptor, either manually or using a GUI tool such as `deploytool`. If the CMP Entity has container-managed relationships, these too must be specified under the `relationships` element of the deployment descriptor.

The `entity` Element

To remind you, the structure of the deployment descriptor is as follows:

```
<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?,
enterprise-beans, relationships?, assembly-descriptor?, ejb-client-jar?)>
```

Here you can see the `relationships` element, with

```
<!ELEMENT enterprise-beans (session | entity | message-driven)+>
```

and the `entity` element defined as

```
<!ELEMENT entity (description?, display-name?, small-icon?, large-icon?,
ejb-name, home?, remote?, local-home?, local?, ejb-class,
persistence-type,
prim-key-class,
reentrant,
cmp-version?, abstract-schema-name?,
cmp-field*, primkey-field?,
env-entry*,
ejb-ref*, ejb-local-ref*,
security-role-ref*, security-identity?,
resource-ref*,
resource-env-ref*,
query*)>
```

Looking first at the `entity` element, much of it will be unchanged. What needs to be specified for a CMP entity are

- `cmp-version` Always set to `2.0`. The value `1.1` is supported only for legacy CMP Entity beans written to the EJB 1.1 specification.
- `abstract-schema-name` Any unique name, this defines the name used to identify the bean in EJB QL queries. It makes sense to base this on the name of the bean. In the case study, the `JobBean` bean has a schema with the name of `Job`.
- `cmp-field` One for each `cmp-field` (but not `cmr-fields`). In the `Job` bean, the `cmp-fields` are `ref`, `customer`, and `description`. The `location` and `skills` fields are `cmr-fields` representing relations to the `Location` and `Skill` beans respectively, and so do not appear.
- `primkey-field` This optional field is used when the `primkey-class` element does not identify a custom primary key class. It is not specified for the `Job` bean, but for the `Location` bean, for example, it is specified and is set to `name`.
- `query` Defines an EJB QL query, associating it with a finder or select method.

Listing 7.12 shows the `entity` element for the `Job` bean.

LISTING 7.12 `Job` Bean's `entity` Element

```
1: <entity>
2:   <display-name>JobBean</display-name>
3:   <ejb-name>JobBean</ejb-name>
4:   <local-home>data.JobLocalHome</local-home>
5:   <local>data.JobLocal</local>
6:   <ejb-class>data.JobBean</ejb-class>
7:   <persistence-type>Container</persistence-type>
```

LISTING 7.12 Continued

```
 8:   <prim-key-class>data.JobPK</prim-key-class>
 9:   <reentrant>False</reentrant>
10:   <cmp-version>2.x</cmp-version>
11:   <abstract-schema-name>Job</abstract-schema-name>
12:   <cmp-field>
13:     <description>no description</description>
14:     <field-name>ref</field-name>
15:   </cmp-field>
16:   <cmp-field>
17:     <description>no description</description>
18:     <field-name>description</field-name>
19:   </cmp-field>
20:   <cmp-field>
21:     <description>no description</description>
22:     <field-name>customer</field-name>
23:   </cmp-field>
24:   <ejb-local-ref>
25:     <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
26:     <ejb-ref-type>Entity</ejb-ref-type>
27:     <local-home>CustomerLocalHome</local-home>
28:     <local>CustomerLocal</local>
29:     <ejb-link>data_entity_ejbs.jar#CustomerBean</ejb-link>
30:   </ejb-local-ref>
31:   <!-- shouldn't be needed, but
          ➥ctx.getEJBHome() returns null in J2EE RI -->
32:   <ejb-local-ref>
33:     <ejb-ref-name>ejb/JobLocal</ejb-ref-name>
34:     <ejb-ref-type>Entity</ejb-ref-type>
35:     <local-home>JobLocalHome</local-home>
36:     <local>JobLocal</local>
37:     <ejb-link>data_entity_ejbs.jar#JobBean</ejb-link>
38:   </ejb-local-ref>
39:   <security-identity>
40:     <description></description>
41:     <use-caller-identity></use-caller-identity>
42:   </security-identity>
43:   <query>
44:     <description></description>
45:     <query-method>
46:       <method-name>findByCustomer</method-name>
47:       <method-params>
48:         <method-param>java.lang.String</method-param>
49:       </method-params>
50:     </query-method>
51:     <ejb-ql>SELECT OBJECT(j)
52: FROM Job AS j
53: WHERE j.customer = ?1</ejb-ql>
54:   </query>
```

LISTING 7.12 Continued

```
55:   <query>
56:     <description></description>
57:     <query-method>
58:       <method-name>findByLocation</method-name>
59:       <method-params>
60:         <method-param>java.lang.String</method-param>
61:       </method-params>
62:     </query-method>
63:     <ejb-ql>SELECT OBJECT(o)
64: FROM Job o
65: WHERE o.location.name = ?1</ejb-ql>
66:   </query>
67: </entity>
```

The definition of the `query` element is as follows:

```
<!ELEMENT query (description?, query-method, result-type-mapping?, ejb-ql)>
```

You can see from the listing that the `query-method` element just identifies the name of the finder or select method. Note that if a finder method is being identified, it is the name appearing in the local-home (or home) interface; that is, without the `ejb` prefix. On the other hand, if a select method is being identified, there will be an `ejb` prefix, because select methods never appear in the interfaces of the bean.

The `result-type-mapping` element applies only if the method identified by `query-method` has identified a select method, and only then if the EJB QL string returns Entity bean references (that is, if the `SELECT [DISTINCT] OBJECT(x)` style of `select_clause` has been used). The allowable values are `Local` and `Remote`, indicating whether the clause should return references through the local or remote interfaces. Obviously, if specified, the bean must provide an interface of the appropriate type; if not specified, the bean must provide a local interface, because this is the implied value for the `result-type-mapping` element.

Of course, all of this deployment information can be entered graphically using the `deploytool`. Figure 7.8 shows some of the equivalent information for Listing 7.12.

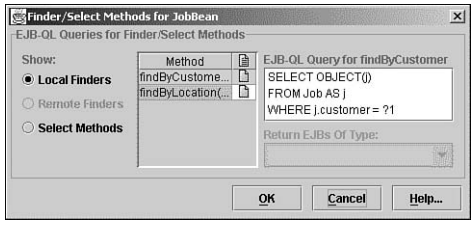


FIGURE 7.8

`deploytool` lets CMP deployment information be defined through a GUI.

The `relationships` Element

The `relationships` element is defined in the EJB 2.0 DTD as follows:

```
<!ELEMENT relationships (description?, ejb-relation+)>
```

That is, it consists of one or more `ejb-relation` elements. These in turn are defined as

```
<!ELEMENT ejb-relation (description?, ejb-relation-name?,
ejb-relationship-role, ejb-relationship-role)>
```

which is a somewhat curious definition: an optional description, an optional name, and then precisely two `ejb-relationship-role` elements. Each of these identifies the role that some bean is playing with respect to the relationship.

It is possible that the same bean appears in both roles to model recursive relationships.

The `ejb-relationship-role` element is defined as follows:

```
<!ELEMENT ejb-relationship-role (description?, ejb-relationship-role-name?,
multiplicity, cascade-delete?, relationship-role-source, cmr-field?)>
```

with

```
<!ELEMENT relationship-role-source (description?, ejb-name)>
```

You can see that the `relationship-role-source` element merely identifies an Entity bean by name. This element's name is perhaps somewhat misleading, because it is neither a "source" nor (for that matter) a target within the relationship. The navigability of the relationship comes from the presence (or not) of the `cmr-field` element. Of course, at least one of the `ejb-relationship-role` elements must have a `cmr-field` specified, and if both do, the relationship is bi-directional.

The choices for the `multiplicity` element are either `One` or `Many`. There will be two such elements in the complete `ejb-relation` element, so this is what gives one-to-one, one-to-many, many-to-one, or many-to-many.

There is also an optional `cascade-delete` element. Perhaps non-intuitively, this is placed on the "child" (multiplicity = many) side of a relationship.

Finally, the `cmr-field` is defined as follows:

```
<!ELEMENT cmr-field (description?, cmr-field-name, cmr-field-type?)>
```

The `cmr-field-name` element just names the `cmr-field` (for example, `skills` or `location` for the `Job` bean). The `cmr-field-type` element is needed only for multi-valued `cmr-fields` (for example, `skills` for the `Job` bean) and indicates whether the return type is a `java.util.Collection` or `java.util.Set`.

Do not confuse this return type with the allowable return types for select methods. These are unrelated areas that just happen to have the same allowable return types.

Okay! Time for an example or two from the case study to see if all that theory makes sense.

Listing 7.13 shows the `ejb-relation` element defining the one-to-many relationship between the `Location` bean and the `Job` bean.

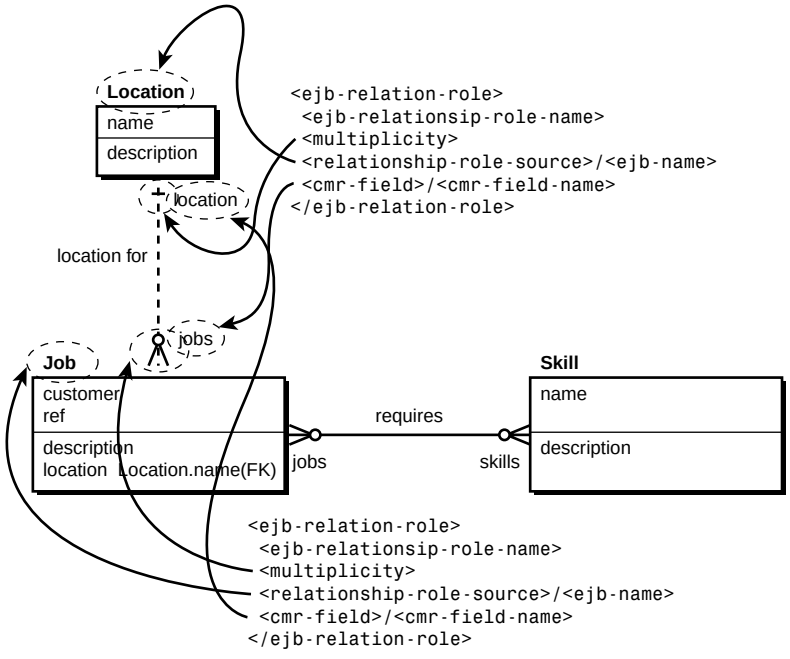
LISTING 7.13 The `Location`/`Job` Relationship

```
 1: <ejb-relation>
 2:     <ejb-relation-name></ejb-relation-name>
 3:     <ejb-relationship-role>
 4:         <ejb-relationship-role-name>JobBean</ejb-relationship-role-name>
 5:         <multiplicity>Many</multiplicity>
 6:         <relationship-role-source>
 7:             <ejb-name>JobBean</ejb-name>
 8:         </relationship-role-source>
 9:         <cmr-field>
10:             <cmr-field-name>location</cmr-field-name>
11:         </cmr-field>
12:     </ejb-relationship-role>
13:     <ejb-relationship-role>
14:         <ejb-relationship-role-name>
            ➥    LocationBean
            ➥</ejb-relationship-role-name>
15:         <multiplicity>One</multiplicity>
16:         <relationship-role-source>
17:             <ejb-name>LocationBean</ejb-name>
18:         </relationship-role-source>
19:         <cmr-field>
20:             <cmr-field-name>jobs</cmr-field-name>
21:             <cmr-field-type>java.util.Collection</cmr-field-type>
22:         </cmr-field>
23:     </ejb-relationship-role>
24: </ejb-relation>
```

Figure 7.9 shows this relationship overlaid onto the implied abstract schema.

FIGURE 7.9

The `ejb-relation` element describes the characteristics of a one-to-many relationship in the abstract schema.



Another example; Listing 7.14 shows the `ejb-relation` element defining the many-to-many relationship between the `Job` bean and the `Skill` bean.

LISTING 7.14 The `Job/Skill` Relationship

```
 1: <ejb-relation>
 2:     <ejb-relation-name></ejb-relation-name>
 3:     <ejb-relationship-role>
 4:         <ejb-relationship-role-name>JobBean</ejb-relationship-role-name>
 5:         <multiplicity>Many</multiplicity>
 6:         <relationship-role-source>
 7:             <ejb-name>JobBean</ejb-name>
 8:         </relationship-role-source>
 9:         <cmr-field>
10:             <cmr-field-name>skills</cmr-field-name>
11:             <cmr-field-type>java.util.Collection</cmr-field-type>
12:         </cmr-field>
13:     </ejb-relationship-role>
14:     <ejb-relationship-role>
15:         <ejb-relationship-role-name>
            ➥    SkillBean
            ➥</ejb-relationship-role-name>
16:         <multiplicity>Many</multiplicity>
```

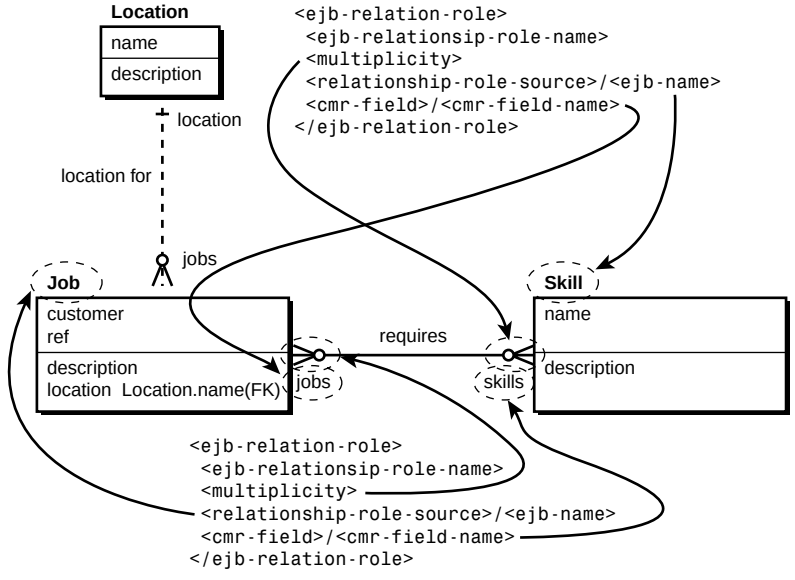
LISTING 7.14 Continued

```
17:         <relationship-role-source>
18:             <ejb-name>SkillBean</ejb-name>
19:         </relationship-role-source>
20:         <cmr-field>
21:             <cmr-field-name>jobs</cmr-field-name>
22:             <cmr-field-type>java.util.Collection</cmr-field-type>
23:         </cmr-field>
24:     </ejb-relationship-role>
25: </ejb-relation>
```

Figure 7.10 shows these elements overlaid onto the abstract schema.

FIGURE 7.10

The `ejb-relation` element describes the characteristics of a many-to-many relationship in the abstract schema.

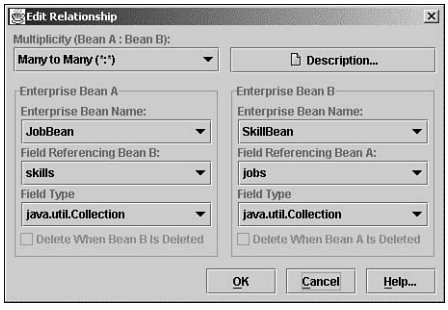


As always, configuring these deployment descriptor elements can be done either by editing the deployment descriptors directly or by using the `deploytool`. Figure 7.11 shows the `deploytool` for the `Job`/`Skill` relationship.

To actually deploy the enterprise application, use the `buildAll` and `deploy` batch scripts in the `day07\build` directory, or use `buildAll` to assemble the enterprise application and `deploy` from `deploytool`.

FIGURE 7.11

`deploytool` can be used to configure relationships through its GUI.



Over the last three days, you have seen Session beans, BMP Entity beans and CMP Entity beans deployed using both the graphical `deploytool` and then using batch scripts. Clearly, the information held in those deployment descriptors is valuable and should be under source code control. Moreover, the mechanism for building and deploying enterprise applications should be able to use that information rather than recreate it from scratch. This suggests that, for the production environment, the graphical `deploytool` should be used only for deploying enterprise applications and for configuring J2EE RI servers.

On the other hand, in the development environment, it can be an error-prone task to attempt to write XML deployment descriptors from scratch. If a valid deployment descriptor already exists, modifying it (to add a new `ejb-ref` element or something similar) can often be accomplished, but larger changes (such as adding a completely new bean) will be more difficult without much practice. Here, `deploytool` comes into its own to modify the enterprise application as required (or indeed, to create an enterprise application from scratch).

When you are happy that the beans and clients in your enterprise application are correctly configured, `deploytool` allows the XML deployment descriptor to be saved, for checking into source code control. This is shown in Figure 7.12.

FIGURE 7.12

`deploytool` allows deployment descriptors to be saved as XML files.

The Tools, Descriptor Viewer menu option brings up a dialog box displaying the XML deployment descriptor; from there, the data can be saved as a file. This menu option is context sensitive, so what it shows will depend on the node selected in the explorer on the left pane in the GUI. The descriptor viewer dialog should be brought up for each node under the enterprise application node, and for the enterprise application node itself. In the case study, this means for each of the clients, for the `data` Entity EJBs, for the `agency` Session EJBs, and for the `agency` application node.

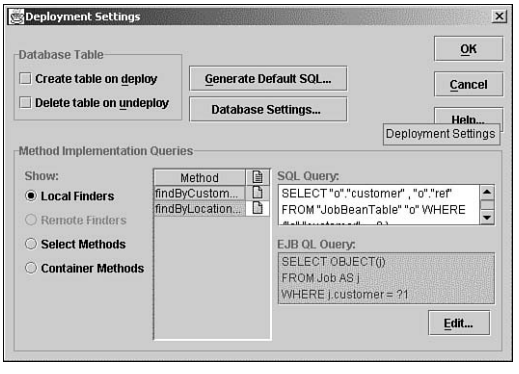
Deploying a CMP Entity Bean

The enterprise application can be deployed either from the command line (`deploy` script in the build directory) or using `deploytool` itself.

However, before an enterprise application containing CMP Entity beans can be deployed, the default SQL must be generated by using the `Deployment Settings` dialog box. This is performed once for each CMP Entity bean, as shown in Figure 7.13.

FIGURE 7.13

`deploytool` allows SQL to create the database schema to be generated.



You can see from the figure that J2EE RI generates default SQL. It allows the SQL query for the finder and select methods to be tuned, and also (the `container methods` radio button) allows the actual SQL to create the tables, insert rows, and so on to be modified also. The case study does *not* change any of this default SQL.

Right at the beginning of today's chapter, it was noted that the schema of the database for the case study had changed from that of Day 6. If one wanted to use the exact schema from Day 6, it could have been entered here.

Also, the dialog allows the underlying tables to be created and deleted on deploy/undeploy. This obviously isn't appropriate for a production environment, because it would delete any data already there. It also is not appropriate for the case study, because there is example data.

> The `CreateCMPAgency` script (provided in the `day07\Database` directory) creates exactly the same schema as that generated by default by J2EE RI. It also populates that schema with the same data as in Day 6 and creates views for backwards compatibility.

You will recall that the auxiliary deployment descriptor `agency_ea-sun-j2ee-ri.xml` contains all the mappings of the logical dependencies of the EJBs to the physical runtime environment. This includes all of the SQL specified in Figure 7.13.

It was noted earlier that when creating a new CMP Entity bean, it is often easiest to load the enterprise application into `deploytool` and then save the XML deployment descriptor using the Tools, Descriptor Viewer menu option/dialog. Unfortunately, `deploytool` does not provide any easy way to write out the auxiliary deployment descriptor, and it is required for the command line approach. The `buildAll` script calls the `addJ2eeRiToEar` script that does precisely this.

The only real option is to save the `agency.ear` file once modified, and then use a tool, such as WinZip, to load up the EAR file. The auxiliary deployment descriptor can be extracted from that.

Patterns and Idioms

This section presents some patterns and idioms that relate to CMP Entity beans. You'll recognize some of the points made here; they were made earlier in the "Container Managed Relationships" section.

Normalize/Denormalize Data in `ejbLoad()`/`ejbStore()`

Under CMP, the `ejbLoad()` and `ejbStore()` methods don't have very much (or indeed anything) to do; the interactions with the persistent data store are done by the EJB container.

However, it may be that the physical schema of the persistent data store (especially if that persistent data store is an RDBMS) does not correspond exactly with the logical schema of the Entity bean.

For example, the `Applicant` table defines two columns—`address1` and `address2`. However, at least conceptually, the `Applicant` Entity bean has a vector field of `address`, of type `String[]`; there could be many lines in the `address` (and it's just that the physical schema of the persistent data store constrains the size of this vector to 2):

```
Santa Claus
No. 1 Grotto Square   (line 1)
Christmas Town        (line 2)
North Pole            (line 3)
The World             (line 4)
Earth                 (line 5)
The Solar System      (line 6, and so on …)
```

Because the `ejbLoad()` method is called after the EJB container has loaded the data, it may renormalize the data. In other words, the data in the two `cmp-field`s of `address1` and `address2` can be converted into the `String[] address` field. The bean's clients' view (as presented by the methods of the local interface) is that the `address` field is a vector.

Conversely, the `ejbStore()` method, called just before the EJB container updates the data, can denormalize the data. In other words, the data in the `address` vector field can be "posted" back to the `address1` and `address2` `cmp-field`s.

Don't Expose `cmp-field`s

Although the EJB specification allows `cmp-field`s to be exposed in the local (or remote) interface of a CMP Entity bean, there are problems with doing so. Because the setter method that corresponds to the field is generated by the EJB container, it is not possible to perform any application-level validation.

Instead, it is better to create a shadow setter method, have it do any validation, and then delegate to the actual `cmp-field` setter method.

You may also want to create a shadow getter method. This would allow you symmetry in the names of the methods, and you could also perhaps do some caching of values or other application-level logic.

As an example, instead of exposing the getter and setter methods for the `description` `cmp-field` of the `Job` bean, you might have a local interface of

```
package data;

import javax.ejb.*;
// imports omitted

public interface JobLocal extends EJBLocalObject {
    String getDescriptionField();
    void setDescriptionField(String description);
```

```
    // code omitted
}
```

with a corresponding implementation of

```
package data;

import javax.ejb.*;
// imports omitted

public abstract class JobBean implements EntityBean {
    public String getDescriptionField() {
        // any application logic here
        return getDescription();
    }
    public void setDescriptionField(String description) {
        // any application logic and validation here
        setDescription(description);
    }
    public abstract String getDescription();
    public abstract void setDescription(String description);
}
```

Don't Expose `cmr-fields`

Although the EJB specification allows `cmr-field`s to be exposed in the local interface of a CMP Entity bean, it may be best not to. There are two reasons why exposing the `cmr-field` causes problems, both related to the returned collection from the getter method of a `cmr-field`:

- The first is that this returned collection is mutable. A client can change of the Entity bean's relationships with other beans by manipulating this collection. In other words, the bean's state is changed without it being aware.
- The second is that the returned collection becomes invalid when the transaction context changes. This is actually good that this is so, but it is a subtle point, and some developers might not appreciate it if less than familiar with EJB transactions (making debugging their application somewhat tricky).

An alternative to exposing the setter method of a `cmr-field` would be for the bean to offer alternative methods, such as `add`*`Xxx`*`()` and `remove`*`Xxx`*`()`, on the bean itself and have these call the setter method. An alternative to exposing the getter method would be to expose a shadow method called something like `get`*`Xxx`*`Copy()`. This would create a copy of the collection. The method name suggests to the client that they will not be able to change the relationships of the bean. Indeed, the returned collection could even be made immutable.

As an example, instead of exposing the getter and setter methods for the `skills` `cmp-field` of the `Job` bean, you might have a local interface of

package data;

```
import javax.ejb.*;
import java.util.*;
// imports omitted

public interface JobLocal extends EJBLocalObject {
    Collection getSkillsCopy();
    void addSkill(SkillLocal skill);
    void removeSkill(SkillLocal skill);
    // code omitted
}
```

with a corresponding implementation of

```
package data;

import javax.ejb.*;
import java.util.*;
// imports omitted

public abstract class JobBean implements EntityBean {
    public Collection getSkillsCopy() {
        List skills = new ArrayList();
        for(Iterator iter = getSkills().iterator(); iter.hasNext(); ) {
            skills.add(iter.next());
        }
        return Collections.unmodifiableList(skills);
    }
    public void addSkill(SkillLocal skill) {
        getSkills().add(skill);
    }
    public void removeSkill(SkillLocal skill) {
        getSkills().remove(skill);
    }
    public abstract Collection getSkills();
    public abstract void setSkills(Collection skills);
    // code omitted
}
```

Enforce Referential Integrity Through the Bean's Interface

Entity beans represent the domain layer in the n-tier architecture, and Entity beans have relationships among themselves. If a method in a bean's interface has an argument of some bean (usually for a relationship), this bean should be defined via the local reference rather than by its primary key. In other words, referential integrity is effectively enforced; the client guarantees that the bean exists, because it passes through an actual reference to that bean.

This idiom is honored implicitly for CMP Entity beans that expose their `cmr-field`s. For CMP Entity beans that provide shadow methods (as discussed earlier), these shadow methods should still deal with local references to the related beans, rather than identifying the related bean by its primary key.

This idiom applies equally to BMP Entity beans. Indeed, if BMP Entity beans are written to follow this idiom, it becomes that much easier to convert them to CMP.

Use Select Methods to Implement Home Methods

Select methods can only be called by a bean itself, so they act as helper methods. A common place where they are often used is in implementing home methods.

For example, the `Job` bean could have provided a home interface to count the number of jobs advertised. This could have been implemented as follows:

```
package data;
import javax.ejb.*;
// imports omitted

public interface JobLocal extends EJBLocalObject {
    int countJobs();
    // code omitted
}
```

with a corresponding implementation of

```
package data;

import javax.ejb.*;
import java.util.*;
// imports omitted

public abstract class JobBean implements EntityBean {
    public int ejbHomeCountJobs() {
        int count = 0;
        for (Iterator iter = ejbSelectAllJobs().iterator(); iter.hasNext(); ) {
            iter.next(); count++;
        }
        return count;
    }
    public abstract Collection ejbSelectAllJobs();
    // code omitted
}
```

The `ejbSelectAllJobs()` EJB QL query string would be simply

```
SELECT OBJECT(j)
FROM Jobs AS j
```

EJB QL does not (yet) support a `SELECT COUNT(*)` syntax, so this is the only way of performing counts (other than resorting to direct access to the data store).

Gotchas

The following is a quick checklist of "gotchas" to help you with your implementation:

- If the primary key is composite, a custom primary key class must be defined. The fields of this class must be `public` and must correspond in name and type to `cmp-field`s of the bean class.
- `cmp-field` and `cmr-field` fields must begin with a lowercase letter, (so that it can be capitalized in the corresponding getter and setter method names).
- If a collection is returned from a `cmr-field`'s getter method, it must not be modified other than through `Iterator.remove()` (see EJB specification, section 10.3.6.1).
- Collections returned by getter methods cannot be used outside of the transaction context in which they were materialized.
- If a bean has no relationships, the `Collection` returned by the `cmr-field`'s getter method will be empty, it will not be `null`. Conversely, if calling a `cmr-field`'s setter method, `null` cannot be used; instead an empty collection (such as `Collections.EMPTY_LIST`) must be passed in.
- EJB QL strings are in single quotes!!!
- EJB QL strings define placeholders as `?1`, `?2`, and so on (rather than the JDBC syntax of just `?`). This allows the arguments of the corresponding finder or select method to be bound in more than once.
- When comparing strings in EJB QL, the strings must be identical to be equal. (This is different from SQL where trailing whitespace is usually ignored.)
- An EJB QL empty string is `''` (0 characters). Some RDBMS treat this as a `null` string, so beware! (See EJB Specification, section 11.2.9.)

Summary

Well done! In three days (Days 5, 6, and 7), you've covered pretty much everything you need to know about writing EJBs. There's a little mopping up of relatively minor issues tomorrow, but you now well and truly have the essentials under your belt.

Today you saw how the n-tier architecture is subtly revised again, with the responsibility for persistence delegated downwards (into subclasses) to the EJB container-generated subclasses of the CMP Entity bean. You also learned that the lifecycle for CMP Entity beans is substantially the same as BMP Entity beans and, in general, there is less coding to be done in the lifecycle methods (`ejbCreate()`, `ejbLoad()`, `ejbStore()`, and `ejbRemove()`).

You now know that `cmp-field`s are defined in the deployment descriptor and correspond to primitive and to `Serializable` objects. They are represented in the CMP bean class as abstract getter and setter methods. You also know that `cmr-field`s work very much the same way but deal with references to the local interfaces of other Entity beans (or collections thereof). This is what makes local interfaces the basis of container-managed relationships.

Finally, you saw how to construct EJB QL queries and how to relate them to both finder methods and to the helper select methods.

Q&A

Q What does CMP stand for?

A CMP stands for Container Managed Persistence.

Q Can the getter and setter methods of `cmp-field`s and `cmr-field`s appear in the bean's interfaces?

A `cmp-field` methods can appear in any interface; `cmr-field` methods can appear only in local interfaces.

Q What keywords or features of ANSI SQL does EJB QL not (yet) support?

A The keywords and features EJB QL doesn't support include `count(*)`, union, group by, order by (among others).

Q How are the parameters of a select method defined?

A The select method is written as a public method in the form `ejbSelect()` in the class itself.

Q How is navigability defined in the deployment descriptor?

A The presence of the `cmr-field` element indicates navigability.

Exercises

The job agency case study defines a complete set of Session beans, and Entity beans. All of the Entity beans except the `Applicant` bean are implemented using CMP; the `Applicant` bean has been implemented using BMP. The exercise is to implement an `Applicant` Entity bean to use CMP. The source code and deployment descriptors can be found under `day07\exercise`, and the directory structure is the same as yesterday.

In more detail, the fields of the `Applicant` bean need to be converted into either `cmp-fields` or `cmr-fields`:

- `login` This is the primary key for the `Applicant` Entity bean and will be a `cmp-field`.
- `name` `cmp-field`
- `email` `cmp-field`
- `summary` `cmp-field`
- `location` `cmr-field` to the `Location` bean. There will be a one-to-many relationship between `Location` to `Applicant`.
- `skills` `cmr-field` to the `Skill` bean. There will be a many-to-many relationship between the `Applicant` and `Skill` beans.

You should find that the `Job` CMP Entity bean acts as a good model for your new version of the `Applicant` bean. One difference is in the primary key. The `Job` bean required a `JobPK` because it had a composite primary key. For your `Applicant` bean, there is no custom primary key, so you will need to nominate the `login` `cmp-field` as its primary key using the `primkey-field` element in the deployment descriptor. Another related difference is that the `Job` bean has to deal with ensuring that its `customer` `cmp-field` is a valid identifier for a `Customer`. Your `Applicant` bean will not have this complication.

There should be no need to change the `Agency` and `Register` Session beans that call the `Applicant` bean, because it is only the internal implementation that is changing, not the interfaces.

The steps you should perform are as follows:

1. If you didn't do so earlier today, convert the `Agency` database to its CMP version. The steps to do this were described in the "A Quick Word about the Case Study Database" section.

2. Re-acquaint yourself with the `ApplicantLocalHome` and `ApplicantLocal` interfaces. However, these will not need to be changed.
3. Update the `ApplicantBean` class; base this on `JobBean`.
4. The CMR relationships that will be built will be bi-directional. This is needed because, otherwise, the code generated by the J2EE RI fails to compile. So, in each of `LocationBean` and `SkillBean`, uncomment the `getApplicants()` and `setApplicants()` methods. Note that these have not been exposed in the respective interfaces.
5. Build the enterprise application (`agency.ear` in the `jar` directory) and then load it into `deploytool`. Delete the `Applicant` bean from the enterprise application, and then add it in again into the `data_entity_ejbs` `ejb-jar`. As you add it, you can specify that it is a CMP bean.
6. Specify the appropriate `cmp-field`s for the new bean.
7. Add a many-to-many relationship from `Applicant` to `Skill`. Make the relationship bi-directional.
8. Add a many-to-one relationship from `Applicant` to `Location`. Make the relationship bi-directional.
9. Use the `deploytool`'s Descriptor Viewer dialog to save the XML deployment descriptor.
10. Now deploy the enterprise application (from the `deploytool` GUI), providing the necessary auxiliary deployment information in the wizard that is displayed. Test your program by using the `AllClients` client, run with `run\runAll`.
11. Optionally, use WinZip or equivalent to extract the auxiliary deployment descriptor and save as `dd\agency_ea-sun-j2ee-ri.xml` from the `agency.ear` file previously generated. Then re-build and deploy using `build\buildAll` and `build\deploy`.

Good luck. A working example can be found in `day07\agency`.

Week 2

Developing J2EE Applications

8 Transactions and Persistence

9 Java Messaging Service

10 Message-Driven Beans

11 JavaMail

12 Servlets

13 JavaServer Pages

14 JSP Tag Libraries

WEEK 2

DAY 8

Transactions and Persistence

You have spent the last three days covering EJBs in detail. In particular, you learned how to specify, implement, configure and deploy container-managed persistence (CMP) Entity beans. Along with BMP Entity beans (Day 6, "Entity EJBs") and Session beans (Day 5, "Session EJBs"), you now have a good appreciation of the EJB technology.

EJBs have been called transactional middle-tier components. Until now, you haven't had to worry too much about transactions in an EJB context because they are reasonably transparent. In fact, you have been using *container-managed transaction demarcation*. However, for those cases where you require explicit control, EJB provides two solutions. You can write beans that manage their own transactions—*bean managed transaction demarcation*—or you can also extend the lifecycle of Session beans to give them visibility of the transaction demarcations. You will be learning about this today.

You spent Day 6 and Day 7, "CMP and EJB QL," comparing the two different persistence approaches offered by EJB in the guise of BMP and CMP Entity beans. The BMP Entity beans were implemented using JDBC, but that is only one of a number of technologies offered by J2EE and Java in general. Today, you will learn

- How to manage transactions explicitly in EJBs
- How transactions are managed "behind the scenes" in an EJB environment
- Some other persistence technologies other than JDBC—specifically, SQLj and JDO

Overview of Transactions

If you've used RDBMS before, or completed a Computer Studies course or read any other J2EE book, you've probably read a section like this one already. But read on anyway, if only to be acquainted with some of the J2EE terminology that is used in this arena.

A *transaction* is an atomic unit of work:

- *Atomic unit* means indivisible—either every step within the transaction completes or none of them do.
- *Work* here usually means some modification to data within a persistent data store. In RDBMS terms, this means one or more `INSERT`, `UPDATE`, or `DELETE`s. However, strictly speaking, it also applies to reading of data through `SELECT` statements.

For a persistent data store to support transactions, it must pass the so-called "ACID" test:

- ***A**tomic*—The transaction is indivisible; the data store must ensure this is true.
- ***C**onsistent*—The data store goes from one consistent point to another. Before the transaction, the data store is consistent; afterwards, it is still consistent.

  The age-old example is of transferring money between bank accounts. This will involve two `UPDATE`s, one decrementing the balance of account #1 and the other incrementing the balance of account #2. If only one `UPDATE` occurs, the transaction is not atomic, and the data store is no longer in a consistent state.
- ***I**solation*—The data store gives the illusion that the transaction is being performed in isolation. Enterprise applications have many concurrent users who are all performing transactions at the same time, so behind the scenes, the data store uses techniques, such as locks, to serialize access to the contended data where necessary.
- ***D**urability*—If a transaction completes, any changes made to the data as a result of that transaction must be durable. In other words, if the power were to fail a millisecond after the transaction has completed, the change must still be there when power is reconnected.

Many data stores use transaction logs to address this requirement. The transaction log holds a journal of changes made to the actual data. When the transaction completes, the log is written to disk, although the actual data need not be.

If you are a Windows user, you may know that Windows NT, 2000 and XP support a filesystem type called NTFS. This replaces the FAT and FAT32 filesystem types used in Windows 95, 98, and ME.

Microsoft often say that NTFS is more reliable, although slightly slower than FAT/FAT32. This is because NTFS has a built-in transaction log, whereas FAT/FAT32 does not.

Many data stores allow transactions to be started explicitly using a syntax such as the following:

```
begin transaction t1
```

where `t1` is the (optional) name transaction. Transactions are completed using either `commit` (make changes permanent) or `rollback` (undo all changes made in the transaction, and revert all data back to the state before the transaction began). Many data stores will use

```
commit transaction t1
```

and

```
rollback transaction t1
```

Some data stores support the concept of *nested transactions*, whereby (for a single user) one transaction can be started while another transaction is still in progress. In other words, two `begin transaction`s can be submitted without a `commit` or `rollback` between them.

However, the EJB specification and many others support only *flat transactions*, whereby one transaction must be completed before another is begun (see EJB specification, section 17.1.2). Consequently, nested transactions are not considered further.

To conclude this short introduction, consider the fragment of SQL shown in Listing 8.1. It transfers $50 from account #20457 to account #19834.

LISTING 8.1 Example Fragment of SQL to Transfer Money Between Accounts

```
 1: begin transaction transfer_money
 2:
 3: update account
 4: set balance = balance - 50
 5: where account_id = 20457
 6:
 7: update account
 8: set balance = balance + 50
 9: where account_id = 19834
10:
11: commit transaction transfer_money
```

In effect, there are two different types of commands:

- Lines 1 and 11 demarcate the transaction.
- Lines 3–5 and 7–9 modify data.

A conventional RDBMS processes all of the SQL in Listing 8.1, but it is performing two different roles in doing so. To understand and process the transaction demarcation commands, it is acting as a *transaction manager*. To understand and process the remaining lines, it is acting as a *resource manager*.

In the EJB specification, these two responsibilities are split. In principle, you can think of the EJB container as acting as the transaction manager, and the persistent data store acting only as the resource manager. The term *transaction coordinator* is sometimes used instead of transaction manager because there could be more than one resource whose transactions are being coordinated.

Splitting the responsibilities of transaction management and resource management has two consequences. For the bean provider, it means that to start a transaction, the bean must interact both with the EJB container and with the persistent data store. The former interaction is largely implicit because it is configured through the deployment descriptor (and as you know, the latter interaction is implicit if CMP Entity beans are used). The other consequence is that, for the persistent data store, it must defer all transaction control responsibility up to the EJB container. Behind the scenes, there is some quite sophisticated communication going on; you'll learn a little about this activity later on today.

Container-Managed Transaction Demarcation

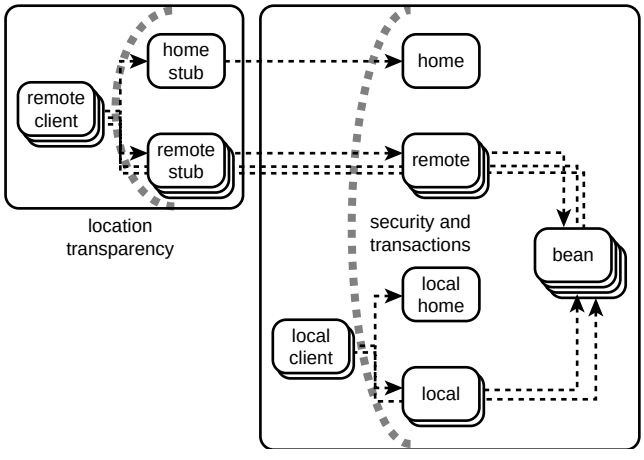
You've spent the last three days writing and deploying EJBs without really having to worry too much about transactions. This isn't to say that there have been no transactions in use; far from it. Every interaction with the database performed in the case study has

involved transactions. However, the Session and Entity beans deployed have used *container manager transaction demarcation* (here referred to as CMTD, though the abbreviation isn't used in the EJB specification). Information in the deployment descriptor indicates when the EJB container should start and commit transactions.

Figure 8.1 shows a diagram that you saw first on Day 6.

FIGURE 8.1

The EJB proxy objects implement transaction (and security) control.



This shows how the EJB proxy objects (those implementing the `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject` interfaces) implement the transaction semantics. This is one of the reasons that a bean must never implement its own remote interface. To do so would mean that it could unwittingly return a reference to itself as a `Remote` interface, subverting any security and transaction checks performed by its proxy.

Listing 8.2 shows a fragment of the deployment descriptor for the `AdvertiseJob` Session bean from Day 7.

LISTING 8.2 Deployment Descriptor for `AdvertiseJob` Session Bean

```
 1: <ejb-jar>
 2:     <display-name>Agency</display-name>
 3:     <enterprise-beans>
 4:         <session>
 5:             <display-name>AdvertiseJobBean</display-name>
 6:             … lines omitted …
 7:             <transaction-type>Container</transaction-type>
 8:             … lines omitted …
 9:         </session>
10:     </enterprise-beans>
11:     <assembly-descriptor>
12:         <container-transaction>
```

LISTING 8.2 Continued

```
13:             <method>
14:                 <ejb-name>AdvertiseJobBean</ejb-name>
15:                     <method-intf>Remote</method-intf>
16:                     <method-name>updateDetails</method-name>
17:                     <method-params>
18:                         <method-param>java.lang.String</method-param>
19:                         <method-param>java.lang.String</method-param>
20:                         <method-param>java.lang.String[]</method-param>
21:                     </method-params>
22:                 </method>
23:             <trans-attribute>Required</trans-attribute>
24:         </container-transaction>
25:         … lines omitted …
26:     </assembly-descriptor>
27: </ejb-jar>
```

As you have seen over the last three days, the `enterprise-beans` element consists of `session` or `entity` elements. The session element has a transaction-type element which, under CMTD, should have the value of Container. For entity elements, the transaction-type is not specified because entity beans must always be deployed using CMTD, so it is implicit.

Each method of the remote interface must be present in a `container-transaction` element. The same is true of the methods in the home interface. The relevant DTD definitions that govern the structure of the deployment descriptor are as follows:

```
<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?,
enterprise-beans, relationships?, assembly-descriptor?, ejb-client-jar?)>
```

The `assembly-descriptor` element is defined as follows:

```
<!ELEMENT assembly-descriptor (security-role*, method-permission*,
                              ➥container-transaction*, exclude-list?)>
```

You can see here that `assembly-descriptor` element is all about providing the information used to create the EJB proxy objects. The `security-role`, `method-permission`, and `exclude-list` elements are security related, and the `container-transaction` element obviously defines transaction-related information.

The security-related elements of the `assembly-descriptor` element are not considered further here. However, the `container-transaction` element is relevant to the discussion. It is defined in the DTD as follows:

```
<!ELEMENT container-transaction (description?, method+, trans-attribute)>
```

and the `method` element is defined as

```
<!ELEMENT method (description?, ejb-name, method-intf?,
                 ➥method-name, method-params?)>
```

So, a `container-transaction` element identifies one or more `methods`. The `method` element identifies a method in the home or remote interface; the `method-intf` element is only needed in those rare occasions when there happens to be a method of the same name in both the home and remote interfaces. The `method-name` must be specified, although the value of `*` can be used as a convenient shortcut to indicate all methods. The `method-params` element is optional and is used to distinguish between overloaded versions of the same method name. If not specified, the `method` element identifies all overloaded versions of the method with the specified name.

Finally, the bit that really matters. The `trans-attribute` element indicates the transactional characteristics to be enforced when the specified method is invoked. A transaction may or may not be in progress; in the terminology of the EJB specification, there may or may not be a current *valid transactional context*. When a method is invoked, the EJB container needs to know what should occur. For example, if there is no transaction in progress, is one needed to execute the method? Should a transaction be started automatically if there isn't one? Perhaps a transaction can be started even *if* another one is in progress? And so on. There are six possible values; their semantics are shown in Table 8.1.

TABLE 8.1 Different CMTD Semantics Are Indicated by the `trans-attribute` Element

`trans-attribute`	*Meaning*	*Notes*
`NotSupported`	Method accesses a resource manager that does not support an external transaction coordinator. Any current transaction context will be suspended for the duration of the method.	The EJB architecture does not specify the transactional semantics of the method.
`Required`	A transaction context is guaranteed. The current transaction context will be used if present; otherwise, one will be created.	Commonly used.
`Supports`	Use valid transaction context if available (acts like `Required`). Otherwise, use unspecified transaction context (acts like `NotSupported`).	Acts as either `Required` or `NotSupported`. This makes the `Supports` a highly dubious choice. The method must guarantee to work in the same way whether or not there is a transaction context available.
`RequiresNew`	A new transaction context will be created. Any existing valid transaction context will be suspended for the duration of the method.	Can reduce contention (for example, for a bean that generates unique IDs or for a bean that writes to a log).

TABLE 8.1 Continued

trans-attribute	*Meaning*	*Notes*
`Mandatory`	A valid transaction context must exist; an exception will be thrown by the EJB container otherwise. The transaction context will be used.	Useful for helper beans' methods, designed to be called only from another bean.
`Never`	There must be no current transaction context. An exception will be thrown by the EJB container otherwise. The method invokes with an unspecified transaction context.	Acts as `NotSupported`.

For Entity beans, only the `Required`, `RequiresNew`, and `Mandatory` `trans-attribute` values are recommended. The problem with `NotSupported`, `Never`, and (if invoked with no current transaction context) `Supports` is that, in addition to performing the business method, the EJB container must also perform the `ejbLoad()` and `ejbStore()` methods. These will be performed with the same transactional context as the business method, which is to say, with *no* transactional context. What might happen then is somewhat undefined, as the EJB specification is at pains to point out. Indeed, it goes so far as to list (in section 17.6.5) four or five different ways in which the EJB container might decide to act.

Never use `NotSupported`, `Never`, or `Supports` as `trans-attribute` values with Entity beans.

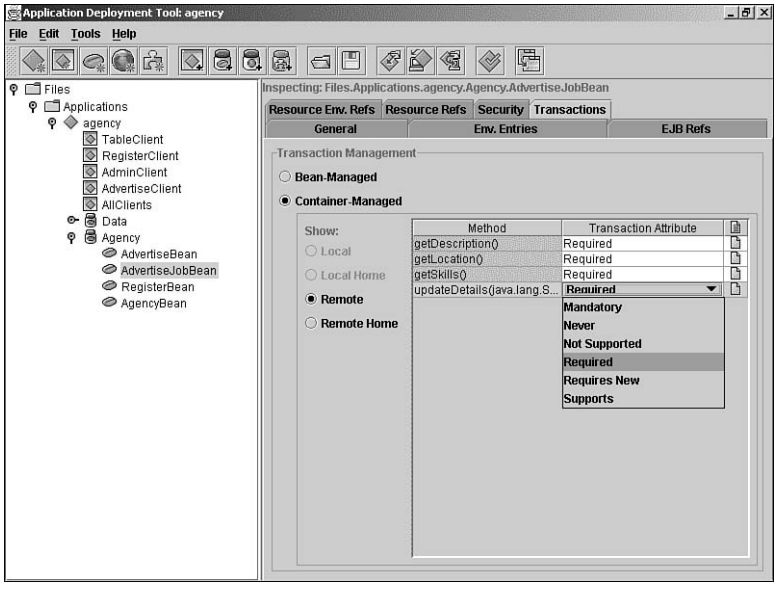
As usual, the `deploytool` GUI can also be used to configure the information within the deployment descriptor, as shown in Figure 8.2.

You are likely to find that the vast majority of your beans' methods will use the `Required` `trans-attribute` value. Indeed, this is the value that has been used in the case study over previous days.

Although CMTD means that the EJB container automatically starts, suspends, and completes transactions, this is not to say that the beans themselves cannot influence the transactions. After all, if an Entity bean hits an error condition that means that the transaction should be aborted, it needs some way to indicate this to the EJB container. As an example, consider the hackneyed example of withdrawing money from a bank account. If the balance would go into the red, (or perhaps more likely, beyond an overdraft limit), the Entity bean that represents the account would want to indicate that the transaction should be aborted.

FIGURE 8.2

`deploytool` lets CMT characteristics be defined on a per-method basis.



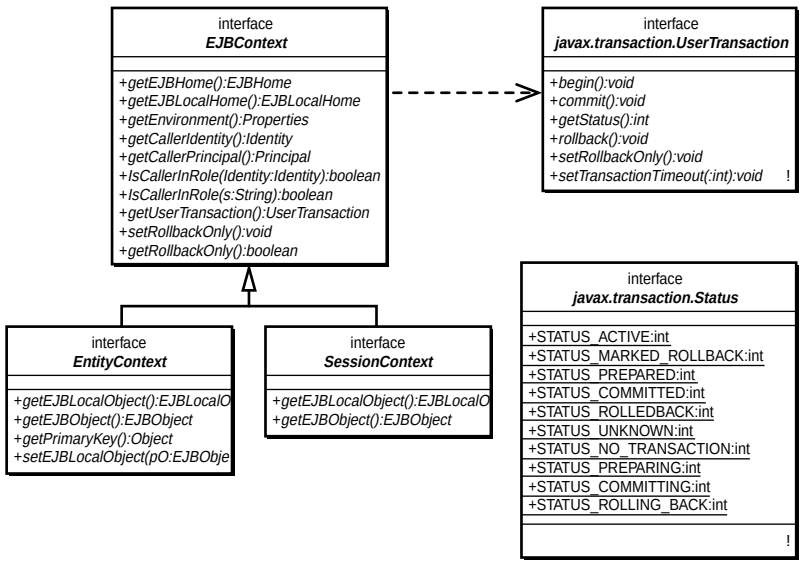
It's interesting to compare the XML deployment descriptor approach to using `deploytool`. The deployment descriptor places the security- and transaction-related information in the `assembly-descriptor` element, away from the definition of the beans themselves (which reside under the `enterprise-beans` element). This is interesting in its own right, because it ties back to the different EJB roles you learned about back on Day 2, "The J2EE Platform and Roles." The intention is that the bean provider completes just the information in the `enterprise-beans` element, while the application assembler completes the information under the `assembly-descriptor`. This allows an EJB component to be used in different applications, with different transaction and security requirements.

In contrast, the `deploytool` does not differentiate between the two roles; the transaction (and security) information are just two of the seven tabs on the right side of the GUI, providing information about the selected EJB; the other five tabs pertain to information found under the `enterprise-beans` element.

To do this, the bean can use two transaction-related methods provided by its `EJBContext`. In the case of a Session bean, this will be the `javax.ejb.SessionContext` passed in through the `setSessionContext()` method, and for an Entity bean, this will be the `javax.ejb.EntityContext` passed in through `setEntityContext()`. To remind you, Figure 8.3 shows a UML class diagram illustrating the methods provided by these interfaces.

FIGURE 8.3

The `EJBContext` provides access to the current transaction.



To cause the transaction to be aborted, the CMTD bean can call `setRollbackOnly()`. This instructs the EJB container to prevent the transaction from being committed. The bean cannot rollback the transaction directly, because the transaction itself is "owned" by the EJB container, not the bean. The `getRollbackOnly()` method obviously just indicates whether the transaction has been marked for rollback.

There is one other transaction-related method in `EJBContext`, namely `getUserTransaction()`. However, this cannot be called by a CMTD bean, and any attempt to do so will result in the EJB container throwing a `java.lang.IllegalStateException`.

The remaining methods in `EJBContext` provide access to the home interface(s) of the bean (`getEJBHome()` and `getEJBLocalHome()`) and to the security context (`getCallerPrincipal()`, `isCallerInRole(String)`). The other methods have been deprecated.

One last point relating to CMTD beans—they must not make use of any resource manager-specific transaction management methods that would interfere with the EJB container's own management of the transaction context. Consequently, a BMP Entity bean cannot call `commit()`, `setAutoCommit()`, and so on a `java.sql.Connection` object.

If your bean does need more fine-grained control over transactions, the bean must be deployed using bean-managed transaction demarcation. This is discussed next.

Bean Managed Transaction Demarcation

If an EJB is deployed using *bean managed transaction demarcation* (here referred to as BMTD, though this abbreviation isn't used in the EJB specification itself), the EJB container allows the bean to obtain a reference to a `javax.transaction.UserTransaction` object using the `EJBContext`. You can see this in Figure 8.3.

Motivation and Restrictions

An EJB might need to be deployed under BMTD if the conditions on which a transaction is started depend on some programmatic condition. It could be that one method starts the transaction and another method completes the transaction.

However, the cases where BMTD is needed are few and far between. Indeed, they are so rare that the EJB specification limits BMTD to Session beans. Entity beans can only be deployed with CMTD. This makes sense because Entity beans represent persistent transactional data; the transaction context should be as deterministic as possible.

Moreover, if a stateless Session bean starts a transaction, it must also commit that transaction before the method completes. After all, the stateless Session bean will have no memory of the client that just invoked its method after that method completes, so one cannot expect that the transaction context is somehow miraculously preserved. If you write a BMTD stateless Session bean that does not commit its own transaction, the EJB container will rollback the transaction *and* throw an exception back to the client (`java.rmi.RemoteException` for remote clients, or `javax.ejb.EJBException` for local).

Indeed, even with stateful Session beans, there is a restriction. Any current transaction in progress when the BMTD Session bean is called will be suspended by the EJB container, not propagated to the BMTD bean. It is possible that, from the bean's perspective, there is a current transaction, but that would refer to any transaction not committed when a method on that bean was last called.

Using the Java Transaction API

When a Session bean is deployed under BMTD, there is an implementation choice as to how it should manage its transactions. If interacting solely with an RDBMS, the Session bean can manage the transactions directly through the JDBC API. Alternatively, it can use the Java Transaction API, defined by the classes and interfaces in the `javax.transaction` and the `javax.transaction.xa` packages. The latter is to be preferred, if only because transactional access to Java Messaging Service resources (you'll be learning more about these tomorrow and the day after) can only be performed through the JTA API. Equally, servlets can also use the JTA API.

The Java 2 Platform Enterprise Edition Specification, the document that defines the interoperability of *all* the technologies that make up the J2EE platform, only discusses transaction interoperability in the context of the JTA API.

If nothing else, the semantics of intermixing JDBC and JTA calls are not exhaustively defined, so this should be avoided to minimize chances of portability problems if moving to a different vendor's EJB container.

For a Session bean to start a transaction, it should first call the `getUserTransaction()` method of its `SessionContext`. You'll recall that this was the method that throws an exception under CMTD, but it is the centerpiece of transaction control under BMTD.

Obtaining a `UserTransaction` does not mean that a transaction has been started. Rather, it must be started using the `begin()` method. The transaction can then be completed using either the `commit()` or the `rollback()` method. The current status can also be obtained using `getStatus()`. This returns an `int` whose meaning is defined by the constants in the `javax.transaction.Status` interface. Some of the most common status values are shown in Table 8.2.

TABLE 8.2 Some of the Constants Defined in `javax.transaction.Status`

Constant	*Meaning*	*Typical actions*
`STATUS_NO_TRANSACTION`	No transaction is active.	`tran.begin()` to start new transaction.
`STATUS_ACTIVE`	A transaction is active and can be used.	Use resource manager. `tran.commit()` to commit `tran.rollback()` to rollback
`STATUS_MARKED_ROLLBACK`	A transaction is active, but has been marked for rollback. Any attempt to commit it will result in a `javax.transaction.RollbackException` being thrown.	`tran.rollback()`

There are more constants in the `Status` interface than those listed in Table 8.2. Later today, (in the "Transactions: Behind the Scenes" section), you'll be learning about some of the "under-the-covers" mechanics of transaction management; the full list is presented there.

Listing 8.3 shows a possible implementation for the `updateDetails()` method of `AdvertiseJob` bean using BMTD.

LISTING 8.3 BMTD Implementation of `AdvertiseJobBean.updateDetails()`

```
 1: package agency;
 2:
 3: import javax.ejb.*;
 4: import javax.transaction.*;
 5: // imports omitted
 6:
 7: public class AdvertiseJobBean extends SessionBean {
 8:     public void updateDetails (String description,
                ➥String locationName, String[] skillNames) {
 9:
10:         int initialTranStatus = beginTransactionIfRequired();
11:
12:         if (skillNames == null) {
13:             skillNames = new String[0];
14:         }
15:         List skillList;
16:         try {
17:             skillList = skillHome.lookup(Arrays.asList(skillNames));
18:         } catch(FinderException ex) {
19:             error("Invalid skill", ex, initialTranStatus);
                                      ➥ // throws an exception
20:             return;
21:         }
22:
23:         LocationLocal location=null;
24:         if (locationName != null) {
25:             try {
26:                 location = locationHome.findByPrimaryKey(locationName);
27:             } catch(FinderException ex) {
28:                 error("Invalid location", ex, initialTranStatus);
                                         ➥ // throws an exception
29:                 return;
30:             }
31:         }
32:
33:         job.setDescription(description);
34:         job.setLocation(location);
35:         job.setSkills(skillList);
36:
37:         completeTransactionIfRequired(initialTranStatus);
38:     }
39:
40:     private int beginTransactionIfRequired() {
41:
42:         UserTransaction tran = this.ctx.getUserTransaction();
43:         // start a new transaction if needed, else just use existing.
44:         // (simulates trans-attribute of REQUIRED)
45:         int initialTranStatus;
```

LISTING 8.3 Continued

```
46:         try {
47:             initialTranStatus = tran.getStatus();
48:             switch(initialTranStatus) {
49:             case Status.STATUS_ACTIVE:
50:                 // just use
51:                 break;
52:             case Status.STATUS_NO_TRANSACTION:
53:                 // create
54:                 try {
55:                     tran.begin();
56:                 } catch(NotSupportedException ex) {
57:         // shouldn't happen (only thrown if asking for nested exception
58:         // and is not supported by the resource manager; not attempting
59:         // to do this here).
60:                     throw new EJBException(
                              ➥ "Unable to begin transaction", ex);
61:                 }
62:                 break;
63:
64:             // code omitted; other Status' covered later
65:
66:             default:
67:                 throw new EJBException(
68:                 "Transaction status invalid, status = " +
                    ➥ statusAsString(initialTranStatus));
69:             }
70:         } catch(SystemException ex) {
71:             throw new EJBException("Unable to begin transaction", ex);
72:         }
73:
74:         return initialTranStatus;
75:     }
76:
77:     /**
78:     * expects initialTranStatus to be either
          ➥ STATUS_NO_TRANSACTION or STATUS_ACTIVE;
79:     * semantics undefined otherwise
80:     */
81:     private void completeTransactionIfRequired(int initialTranStatus) {
82:
83:         UserTransaction tran = this.ctx.getUserTransaction();
84:
85:         // if transaction was started, then commit / rollback as needed.
86:         // (simulates trans-attribute of REQUIRED)
87:         if (initialTranStatus == Status.STATUS_NO_TRANSACTION) {
88:             try {
89:                 if (tran.getStatus() == Status.STATUS_MARKED_ROLLBACK) {
90:                     tran.rollback();
```

LISTING 8.3 Continued

```
91:                 } else {
92:                     tran.commit();
93:                 }
94:             } catch(Exception ex) {
95:                 throw new EJBException(
                        ➥ "Unable to complete transaction", ex);
96:             }
97:         }
98:     }
99: }
```

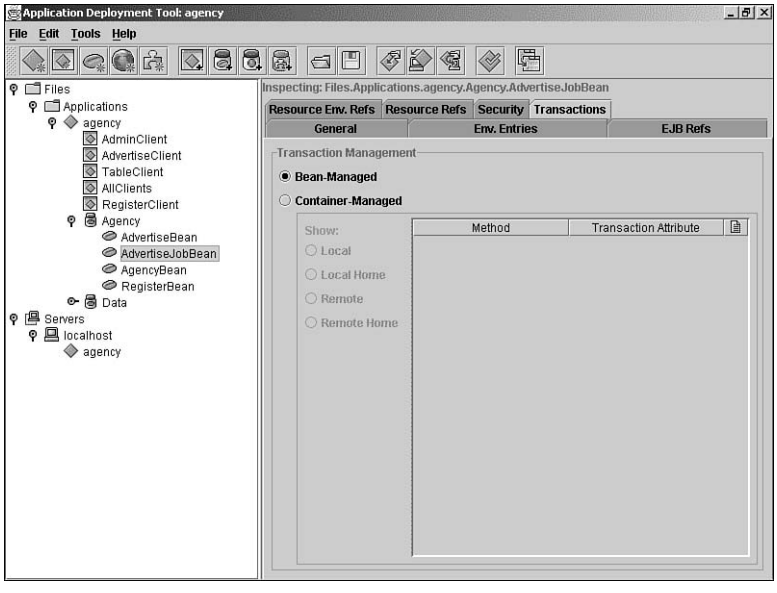
The two helper methods, `beginTransactionIfRequired()` and `completeTransactionIfRequired()`, isolate the actual transaction management code, so it can be reused across different methods.

Deploying a BMTD Bean

Of course, when deploying a bean under BMTD, the deployment descriptor should indicate a `transaction-type` element of `Bean`, and you will not need any container-transaction elements under the `application-assembly` element. Figure 8.4 shows `deploytool` for the `AdvertiseJob` bean, indicating this fact.

FIGURE 8.4

BMTD is indicated through the deployment descriptor, as shown in `deploytool`.



Incidentally, if a BMTD Session bean calls `getRollbackOnly()` or `setRollbackOnly()` on its `SessionContext`, the EJB container will throw a `java.lang.IllegalStateException`. This is reasonable; if a BMTD has access to the `UserTransaction` object, it has no need for these methods. Instead, it can call the `getStatus()` method of `UserTransaction`, and explicitly call `rollback()` if needed.

Client-Demarcated Transactions

As well as Session beans managing their own transactions, it is also possible for clients to initiate the transaction and have it propagate through to the EJBs. Here, "client" means either an application client written using the Swing GUI (such as you have seen in the case study), or it could equally refer to a Web-based client implemented using servlets and JSPs.

For either of these clients, the EJB architecture requires that a `UserTransaction` context can be obtained via JNDI, bound under the name of `java:comp/UserTransaction`. So the code fragment shown in Listing 8.4 will do the trick.

LISTING 8.4 Obtaining a `UserTransaction` Object from JNDI

```
1: // assuming:
2: // import javax.naming.*;
3: // import javax.transaction.*;
4: InitialContext ctx = new InitialContext();
5: UserTransaction tran = (UserTransaction)
   ➥ctx.lookup("java:comp/UserTransaction");
6: tran.begin();
7: // call session and entity beans
8: tran.commit();
```

That said, if you find yourself needing to use client-demarcated transactions, you should look at your application design and see if you are happy with it. After all, Session beans (are meant to) represent the application logic of your application, and this should surely include defining the transactional boundaries of changes to persistent data. Application clients should only provide a presentational interface to your application.

If that philosophical argument does not appeal, perhaps this might. A rogue client could be coded such that it begins a transaction, interacts with (and therefore ties up) various resources, such as Entity beans, and then not commit. This could seriously impact the performance of your application.

Exceptions Revisited

On Days 5 and 6, you learned the appropriate exceptions for your EJB to throw. In summary

- To throw an application-level exception (indicating that a possibly recoverable condition has arisen), throw any checked exception (excluding `java.rmi.RemoteException`).
- To throw a system-level exception (indicating that a non-recoverable severe condition has arisen), throw any `java.lang.RuntimeException` (usually a subclass of `javax.ejb.EJBException`).

If an application-level exception is thrown by a bean, it is up to that bean whether the current transaction is affected or not. If the bean takes no action other than raising its exception, the current transaction will be unaffected. The exception will simply propagate back to the calling client.

However, CMTD beans may decide to mark the current transaction for rollback, meaning that the "owner" of the transaction (the EJB container or some BMTD bean) will be unable to commit that transaction.

If a BMTD bean hits an error condition, it has a choice. Because it "owns" the transaction, it can simply do a rollback. Alternatively, it might elect to keep the transaction active.

If a system-level exception is thrown by a bean, this *does* have consequences for any current transaction. If any bean throws a system exception, the EJB container will mark the current transaction for rollback. If that bean happens to be a CMTD bean, and the EJB container started a transaction just before invoking the CMTD method (as a result of a `Required` or `RequiresNew` `trans-attribute`), the EJB container will actually rollback that transaction.

To summarize,

- An application-level exception may or may not leave the current transaction active; use `getStatus()` or `getRollbackOnly()` to find out.
- A system-level exception will either mark the current transaction for rollback or even do the rollback.

One last thing on transactions and exceptions. Most of the exceptions in `javax.ejb` (`CreateException`, `RemoveException`, and so on) are application exceptions. Some of these, especially with CMP Entity beans, are raised by the EJB container itself. Rather unhappily, the EJB specification does not mandate whether these application exceptions should mark any current transaction for rollback (see section 10.5.8). Instead, it just indicates that the `getStatus()` or `getRollbackOnly()` methods should be used to determine the status of the current transaction. In practical terms, what this means is that different EJB containers could have different implementations, compromising portability.

Extended Stateful Session Bean Lifecycle

Occasionally, there is a need for a stateful Session bean to have visibility as to the progress of the underlying transaction. Specifically, a bean might need to know

- Has the transaction started?
- Is the transaction about to complete?
- Did the transaction complete successfully or was it rolled back?

For example, consider the age-old example of a `ShoppingCart` bean. In its `purchase()` method, it is likely to modify its internal state. Perhaps it holds its contents in a `java.util.List` called `currentContents`. On `purchase()`, it might move the contents of `currentContents` to another `java.util.List` called `recentlyBought`.

Suppose, then, that the transaction that actually modifies the persistent data store fails to complete. Maybe the shopper doesn't have enough limit left on his or her credit card. Because Session beans are not transactional, the `ShoppingCart` bean needs to know that it should reset its internal state back to the beginning of the transaction. In other words, it needs to move the contents of the `recentlyBought` list back over to `currentContents`.

Obviously, there is no issue for stateful Session beans deployed under BMTD, because they are the owner of the transaction anyway and know when the `tran.begin()` and `tran.commit()` methods will be invoked. But for CMTD Session beans, there is an issue.

The EJB specification addresses this by extending the lifecycle of the bean. If a stateful Session bean implements the `javax.ejb.SessionSynchronization` interface, three additional lifecycle methods are defined and will be called at the appropriate points:

- `afterBegin()` The transaction has just begun.
- `beforeCompletion()` The transaction is about to be committed.
- `afterCompletion(boolean)` The transaction has completed. The `boolean` argument has the value `true` to indicate that the transaction was committed, or `false` to indicate that the transaction was rolled back.

Figure 8.5 is a reworking of Figure 5.14 that you saw back on Day 5. It shows the stateful Session bean's view of its lifecycle. (The client's view and the actual lifecycle managed by the EJB container are unchanged).

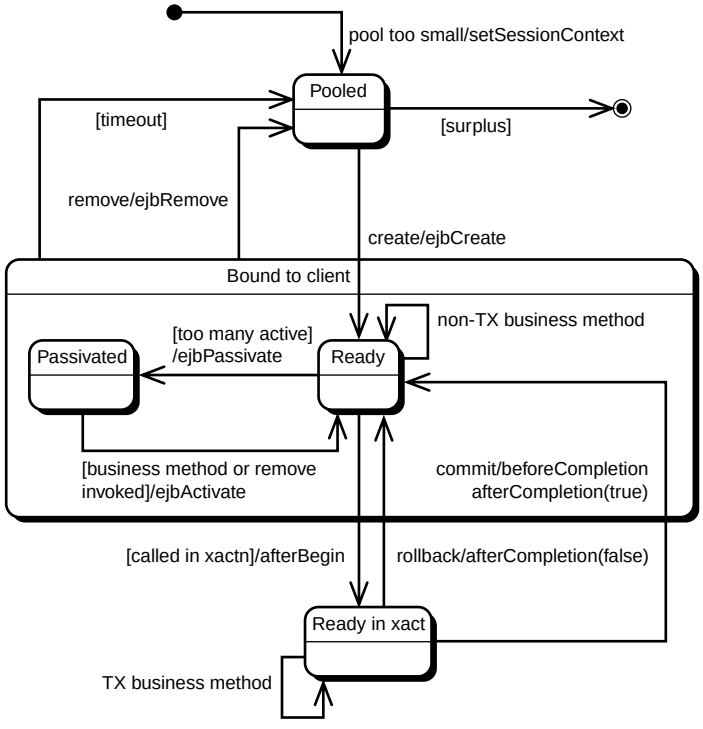


FIGURE 8.5

The `Session Synchronization` interface gives the stateful Session bean visibility to the transactions managed under CMTD.

One common pattern for using this interface is to use the `afterBegin()` method to load any data from the data store (perhaps in the form of Entity beans), and then use the `beforeCompletion()` method to write any cached data that may have changed. One immediate application might be with respect to multi-valued `cmr-field`s. You will recall from yesterday that collections returned by the getter method of a `cmr-field` are valid only for the duration of a transaction. These two methods scope the duration that such a returned collection can be used.

There are analogies here also with SQL, where the `afterBegin()` corresponds to a `SELECT … WITH HOLDLOCK` statement, and `beforeCompletion()` corresponds to the `UPDATE` statements.

If the `SessionSynchronization` interface is implemented by a Session bean, the only allowable values for the `trans-attribute` element in its deployment descriptor are `Required`, `RequiresNew`, or `Mandatory`. This is because these are the only attributes that can guarantee the presence of a transaction.

Transactions: Behind the Scenes

The EJB specification starts off by listing nine goals. The second of these reads as follows:

> *"The Enterprise JavaBeans architecture will make it easy to write applications: Application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, and other complex low-level APIs."*

The EJB specification (downloadable from `http://java.sun.com/products/ejb/index.html`) largely succeeds in addressing this goal—as a developer, you really do not need much knowledge about how transactions are managed. The fact that this book only covers transactions today is testament to that.

Nevertheless, like most technical topics, it can be helpful to have an insight as to what is going on "behind the scenes." But if you want to skip this material and make a shorter day of it, please do so. You can always re-read at a later date if you find yourself wanting to know more.

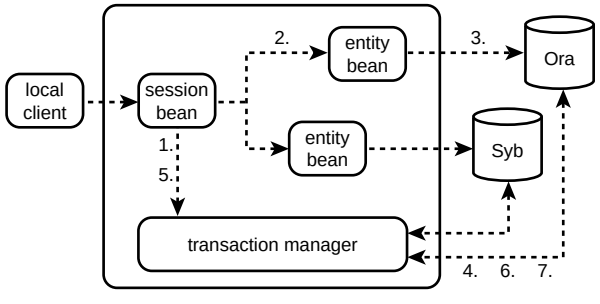
Transaction Managers, Resource Managers, and 2PC

You already know about the terms resource manager and transaction manager (or transaction coordinator). In most EJB applications, an RDBMS will take the place of a resource manager, and the EJB container itself will be the transaction manager.

This division of responsibilities is required because in an EJB architecture, the data used by Session beans or represented by Entity beans may reside in more than one persistent data store. For example, one Entity bean might map onto an Oracle RDBMS, and another Entity bean may map onto a Sybase RDBMS, as shown in Figure 8.6. (The numbers will be explained shortly.)

FIGURE 8.6

The J2EE platform separates resources and transaction managers.



Leaving the transaction management responsibilities with the RDBMSs is not suitable. Doing so would mean that each RDBMS would have its own transaction. If the first transaction succeeded but the second transaction failed, the logical data set represented by the Entity beans would no longer be consistent. The "C" of the ACID test would be broken.

Another case where only a single transaction is required is when interacting with JMS queues or topics. You can imagine that a queue might implement a To Do list. A task on the To Do list might be "invoice customer A for $20," involving an update to an RDBMS. If the task is removed from the To Do list as one transaction, and the update to the RDBMS performed as another transaction, there is again the possibility that the second transaction fails. In other words, the task is removed from the To Do list, but no invoice is raised.

The *two phase commit* protocol (more commonly called just 2PC) is the mechanism by which the transaction manager (EJB container) interacts with each of the resource managers (RDBMS or JMS queues and topics). In the EJB environment, it works as follows (the numbers correspond to the steps in Figure 8.6):

1. The transaction manager within the EJB container creates a new transaction as needed. Generally, this will be when a Session bean's method is invoked.

   If the Session bean has been deployed under CMTD, the bean's proxy will make this request to the transaction manager. If the Session bean is deployed under BMTD, the bean itself will effectively make this request.
2. The Session bean interacts with Entity beans. The current transaction will be propagated through to them (assuming they are deployed with `Required` or `Mandatory` value for their `trans-attribute` element).
3. In turn, the Entity beans interact with the RDBMSs through an XA-compliant `java.sql.Connection`. "XA-compliant" means supporting the two phase commit protocol (or 2PC); more on this shortly. If the Entity bean is BMP, the interaction with the RDBMS will be done by the bean itself; if the bean is CMP, the interaction will be by the generated subclass. Either way, it amounts to the same thing.
4. Each of the XA-compliant `Connections` registers itself with the EJB's transaction manager. More correctly, a `javax.transaction.xa.XAResource` representing and associated with the `Connection` is registered as part of the current `javax.transaction.Transaction`.
5. When all method calls to the Entity beans have been made, the Session bean indicates to the transaction manager that the transaction should be committed.

6. The transaction manager performs the first "prepare" phase of the commit. It iterates over each of the `XAResources` that constitute the transaction and requests confirmation that they are ready to be committed. In turn, the `XAResource` just delegates this request to its corresponding XA-compliant `java.sql.Connection`.
7. When all resources have indicated that they are prepared, the transaction manager performs the second "commit" phase. It again iterates over each of the `XAResources` and requests them to commit.

The JTA API

An XA-compliant `java.sql.Connection`, as previously described, is one that provides the ability to return an `XAResource` for registering with the current transaction. The XA protocol is an industry standard, defined by the X/Open group, to allow a transactional resource manager to participate in a global transaction controlled by an external transaction manager. The JTA API is effectively a mapping into Java of this XA protocol.

In fact, the `java.sql.Connection` interface does not mandate XA-compliance, so the previous description was a slight simplification. Instead, XA-compliance is provided by classes and interfaces in the `javax.sql` package, part of the J2EE platform. Some of the more relevant classes of `java.sql`, `javax.sql`, `javax.transaction`, and `javax.transaction.xa` are shown in Figure 8.7.

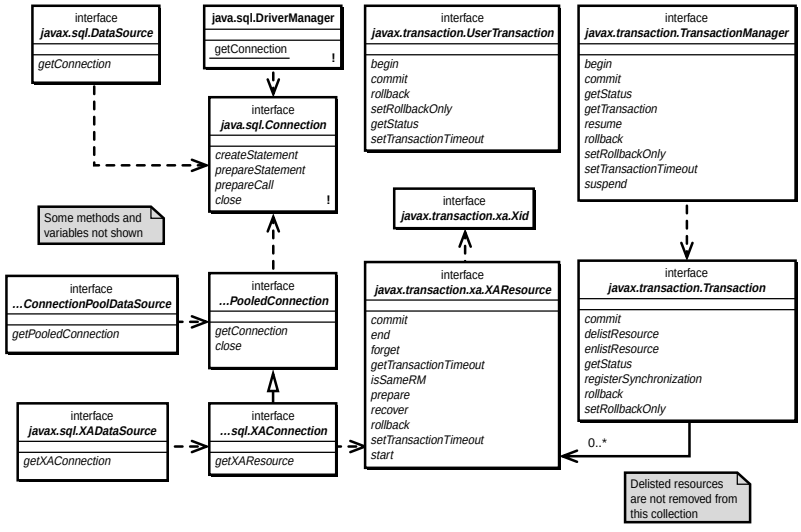


FIGURE 8.7

The `javax.sql` and `javax.transaction` packages together provide support for 2PC against RDBMSs.

As you know, the `java.sql.Connection` interface represents a connection to an RDBMS. In J2SE applications, `java.sql.DriverManager` can be used to create a connection. Under J2EE, a `javax.sql.DataSource` object is used.

In J2EE enterprise applications, reusing connections through some sort of connection pool is critical to ensuring performance and scalability. Many implementations of `DataSource` provide built-in connection pooling. The connection returned by `DataSource.getConnection()` is effectively a logical connection temporarily associated with an underlying physical connection. When the `close()` method on the logical connection called, the underlying physical connection is not closed but is, instead, returned to a connection pool.

J2EE also offers another approach for RDBMS vendors to provide connection pooling, through the `javax.sql.ConnectionPoolDataSource` interface. This returns `PooledConnections` that are physical connections to the database. In turn, they provide a `getConnection()` method that returns a logical connection that wraps them, so the final effect is much the same as before.

If you want, you can think of the `getConnection()` method of `PooledConnection` as leasing the connection from the pool. The `close()` method releases the `PooledConnection` back into the pool to be used again.

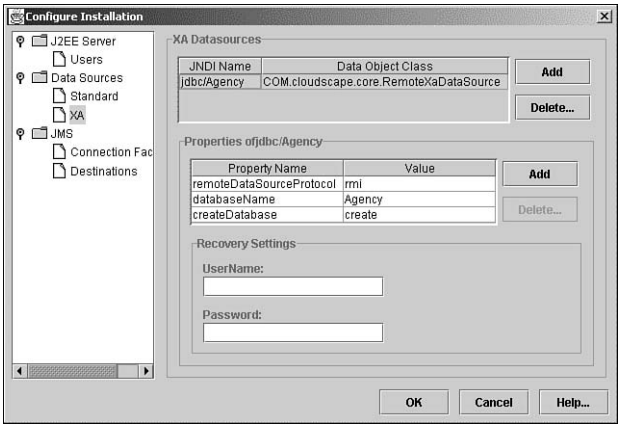
Closely related to `ConnectionPoolDataSource` is the `javax.sql.XADataSource`, which returns `javax.sql.XAConnections`. This is a sub-interface of `PooledConnection`, so it works in the same way, providing the `getConnection()` to return a logical `java.sql.Connection` that wraps it. However, it also provides the `getXAResource()` method that returns a `javax.transaction.xa.XAResource`. Consequently, the `XAConnection` acts as a bridge between the resource manager's notion of connection and the transaction manager's notion of resource.

A J2EE-compliant resource manager must be able to support each of these three different `DataSource` interfaces (`XADataSource`, `ConnectionPoolDataSource`, and `DataSource` itself).

The J2EE RI `deploytool` can be used to configure `XADataSources` against Cloudscape by using the `Tools, Server Configuration` menu option. Alternatively, the `resource.properties` file under `%J2EE_HOME%\config` can be edited directly. The dialog box and required entries are shown in Figure 8.8. (You'll also need to remove the previous definition of `jdbc/Agency`, listed under `DataSources/Standard` node of the Explorer).

FIGURE 8.8

deploytool *can be used to configure* `XADataSources`.



As you saw earlier today, the EJB application's interface into the EJB container's transaction manager implementation is solely through the `javax.transaction.UserTransaction` interface. Indeed, there is no direct way to access the `javax.transaction.TransactionManager` object or the `javax.transaction.Transaction` object that corresponds to the `UserTransaction` (though obviously they are available to the EJB container itself).

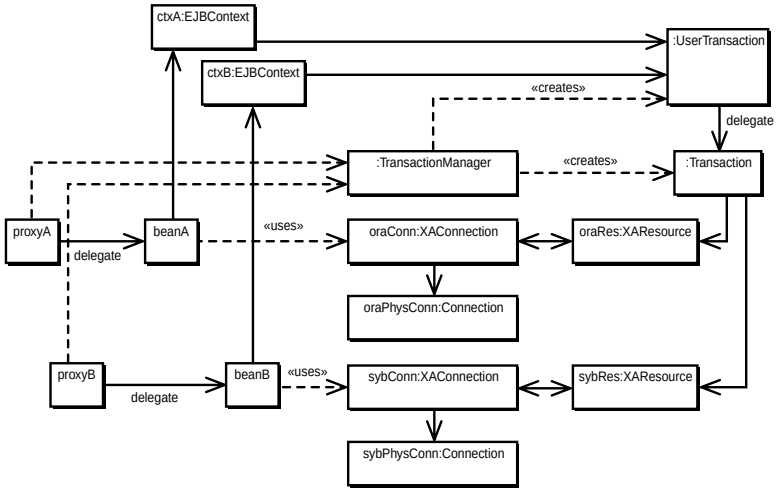
Some EJB containers make the `Transaction` and `TransactionManager` interfaces available—through JNDI. As ever, using these value-add features will compromise application portability.

Figure 8.9 shows an object instance diagram to indicate (some of) the objects that might be instantiated to represent the scenario shown back in Figure 8.6.

In practice, there probably would not be two objects for the `XAConnection` and `XAResource` interfaces (`oraConn`/`oraRes` and `sybConn`/`sybRes` in the diagram). More likely, an RDBMS vendor would implement a concrete class that implements both of these two interfaces. The JTA API describes such a class as the `ResourceAdapter`. Consequently, each transaction managed by the EJB container's transaction manager will have a collection of `ResourceAdapters`, each also being a physical connection to some resource manager. Indeed, if you have used the Adapter design pattern, you'll recognize that the `ResourceAdapter` is well named, combining two orthogonal interfaces into a single instance.

FIGURE 8.9

An object instance diagram showing `XAConnections` *and* `XAResources`.



What If It Goes Wrong?

You've seen how the 2PC protocol is intended to work. However, the whole point of 2PC is to ensure transactional consistency, even in the event of an unexpected failure. So, what happens when it goes wrong?

There are two cases to deal with. First, it could be that a resource manager enlisted into the transaction may no longer be available when the application (in an EJB context, the Session bean or its proxy) decides to commit. It could be that the network has failed since the original interaction with the Entity bean that represents some data residing on that resource manager.

In this first case, the prepare phase of the 2PC protocol fails. Because the transaction manager has been unable to get an acknowledgement within its timeout, it will roll back the transaction. When the resource manager that failed is restarted, it will (as part of its so-called recovery process) automatically roll back any work done as a part of the transaction.

In the second case, a resource manager becomes unavailable after it has acknowledged the prepare, but before the commit phase. This rare case causes more problems because the transaction manager may already have sent the commit message to some other resource managers. Nevertheless, the transaction manager will continue to send the commit message to all other resource managers.

When the resource manager that failed is restarted, it will detect that it had acknowledged a prepare. It then contacts the transaction manager to determine whether the transaction was actually committed or was rolled back. It then performs the same (commit or rollback) as part of its recovery process.

The prepare phase is more than the transaction manager checking that all resource managers are still available. It is also possible for resource managers to unilaterally decide to abort a transaction for some reason. When this happens, the transaction manager will send a rollback message to all participating resource managers, rather than a commit.

If you have done any JavaBean programming, some of this will be starting to sound familiar. The `java.beans.VetoableChangeSupport` class works in a very similar way.

In addition to failures of the prepare or the commit phase of the 2PC, there are also occasional cases when a resource manager may take a so-called heuristic decision that could be in conflict with the semantics of the transaction. For example, a resource manager could have a policy that, once prepared, it will commit if the transaction manager has not confirmed the outcome (commit or rollback) within a certain period. One reason that a resource manager might do this would be to free up resources.

If a heuristic decision is made, it is the responsibility of the resource manager to remember this decision. In an RDBMS, this is often stored in some sort of system table. Put bluntly, this information is required to allow the administrator to correct any issues with the data if the heuristic decision went against the transaction's actual outcome. You might have noted that the `XAResource` interface defines a `forget()` method; this allows the resource manager to finally forget that a heuristic decision was made.

This probably all sounds pretty arcane, but is needed if you want to understand the full set of status values defined by the `javax.transaction.Status` interface. You'll recall that a subset of these was presented in Table 8.2. Table 8.3 shows all of the constants.

TABLE 8.3 All of the Constants Defined in `javax.transaction.Status`

Constant	*Meaning*	*Typical Actions*
`STATUS_NO_TRANSACTION`	No transaction is active.	`tran.begin()` to start new transaction.
`STATUS_ACTIVE`	A transaction is active and can be used.	Use resource manager. `tran.commit()` to commit. `tran.rollback()` to rollback.
`STATUS_MARKED_ROLLBACK`	A transaction is active, but has been marked for rollback. Any attempt to commit it will result in a `javax.transaction.RollbackException` being thrown.	`tran.rollback()`

TABLE 8.3 Continued

Constant	*Meaning*	*Typical Actions*
`STATUS_PREPARED`	A transaction is active, and is in the process of being committed. The first "prepare" phase of the 2PC protocol has completed.	Nothing; wait for transaction to complete.
`STATUS_PREPARING`	A transaction is active, and is in the process of being committed. The first "prepare" phase of the 2PC protocol is in progress.	
`STATUS_COMMITTING`	A transaction is active, and is in the process of being committed. The second "commit" phase of the 2PC protocol is in progress.	
`STATUS_ROLLING_BACK`	A transaction is active, and is in the process of being rolled back.	
`STATUS_COMMITTED`	The previous transaction has committed, but there is likely to be some heuristic data available (otherwise, the status returned would have been `STATUS_NO_TRANSACTION`).	Use administrative tool to forget heuristics after checking data is valid in resource. `tran.begin()` to start new transaction, but heuristic data may be lost.
`STATUS_ROLLEDBACK`	The previous transaction has rolled back, but there is likely to be some heuristic data available (otherwise, the status returned would have been `STATUS_NO_TRANSACTION`).	
`STATUS_UNKNOWN`	This is a transient status. Subsequent calls will resolve to another status.	Wait.

JTA Versus JTS

There are two Java APIs relating to transactions:

- *The Java Transaction API (JTA)*—(Already introduced) The JTA classes and interfaces reside in the `javax.transaction` and `javax.transaction.xa` packages.
- *The Java Transaction Services API (JTS)*—This is a Java mapping for the OMG's Object Transaction Service v1.1 to interoperate with CORBA ORB/TS standard interfaces. This includes on-the-wire propagation of transactions over CORBA's IIOP network protocol. The JTS classes and interfaces reside in the `javax.jts` package.

For its part, the JTS specification (downloadable from `http://java.sun.com/products/jts/index.html`) mandates that any compliant Transaction Manager implementation must also provide complete support for the JTA API, so you can think of JTS as the "back-end" of JTA. Some Java APIs (such as JDBC and JNDI) use the terms API and SPI, where API is the application developer's programming interface, and SPI is supporting the service-provider's interface. For example, a JDBC driver written by Oracle Corp. would be an implementation of the JDBC SPI. Using this terminology, JTS is the SPI portion of JTA.

The EJB specification mandates that the EJB container must provide access to the JTA API, so some vendors will do this by implementing JTS. Such vendors can then offer transaction propagation between EJBs that reside on different EJB servers, because both such servers will effectively appear as CORBA servers implementing OTS. Figure 8.10 shows this.

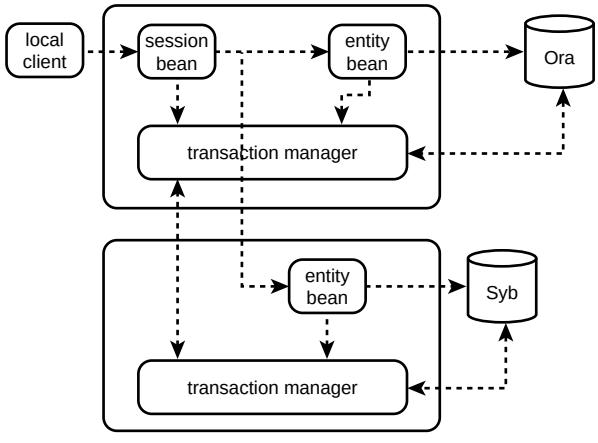


FIGURE 8.10
JTS support means transactions can be propagated between EJB servers.

Obviously, both EJB servers need to support JTS, but they need not be implemented by the same EJB container vendor. The transaction manager on the called EJB server is enlisted as a resource of the transaction manager on the calling EJB server.

While the EJB specification mandates JTA support, it does not mandate that JTS be used to realize this support. In other words, the EJB container is free to provide any implementation of the interfaces in JTA, but it need not support the additional requirements (principally CORBA interoperability) that make up the JTS API. Some vendors—perhaps those from a database or a Web application background—may implement JTA entirely "within" their EJB container and offer no CORBA interoperability services. On the other hand, the EJB specification also indicates that if an EJB container vendor does elect to provide transactional propagation, it must do so by supporting OTS.

In fact, if the vendor elects to support transactional propagation, the EJB specification requires support for OTS v1.2 (see for example section 19.6.1.1). Strictly speaking, JTS 1.0 is a Java mapping only for OTS v1.1, so JTS 1.0 support does not in itself fully address the requirements of the EJB specification.

If the area of transaction propagation is of particular interest to you, you should make sure that your EJB container vendor can clarify its position to you.

Some EJB container vendors will also be vendors of CORBA products, and so are likely to implement JTA just by implementing JTS.

Overview of Persistence Technologies

The second main topic for today is to discuss persistence options and look at some of the Java technologies available in this space. At this point, you might be thinking that this is a moot point; after all, you learned yesterday and on Day 6 how to develop Entity beans, so what else is there to address?

In response to that question, consider the following two points. First, you used JDBC to implement BMP Entity beans on Day 6. However, there are other technologies that may involve less work and could even lend themselves to code generation, or indeed, support persistence transparently. Second, many J2EE applications will not use EJBs. There are many successful J2EE enterprise applications built only with servlets, JSPs, and data access code. Moreover, commercial EJB containers can be costly to purchase, and this might also be a consideration for your organization.

The three technologies that you will learn more about are as follows:

- *JDBC*—This is the most mature of the Java persistence technologies. There have been multiple versions over the last few years (sometimes renamed along the way), which can be confusing.

  The JDBC API itself is not covered, because it is presumed that you are or have become familiar with it.

- *SQLj*—This is actually three specifications that combine Java and SQL, either client-side or within the RDBMS. Some aspects of SQLj are supported through JDBC (v2.0 and later).

  At the time of writing, SQLj was being developed by a consortium of companies that includes Sun, Oracle, IBM, and Sybase. You can learn more at `http://www.sqlj.org`.

- *Java Data Objects (JDO)*—This aims to make persistence transparent, either for small embedded applications or large scale enterprise applications. This latter objective means it can either replace or supplement Entity EJBs. (At the time of writing, this specification was still in draft.)

  At the time of writing, JDO was being developed through the Java Community Process, JSR-000012. You can learn more at `http://access1.sun.com/jdo`. You can learn about its relationship with JDBC at `http://java.sun.com/products/jdbc/related.html`.

This is by no means a definitive list. Specifically, there is nothing to prevent you from using a vendor-specific API to persist your data, as is used by most OODBMS vendors. While OODBMSs are not as mainstream nor as prevalent as RDBMSs, they have many advocates who argue vociferously that the best place to store objects is in an object-based database. Examples of OODBMS products include (in no particular order) O2, Objectivity, ObjectStore, Versant, FastObjects (previously POET), Persistence, Jasmine, Gemstone, and ozone. From a J2EE perspective, many of the OODBMS vendors have re-branded their products to be EJB application servers, while others have positioned their technology to provide a simple way to implement persistence (within a BMP Entity bean).

Yet another alternative is to use an object/relational mapping product. In a sense, these tools combine the familiarity of RDBMS with the intuitiveness of objects. The information that maps the object instances to the relational schema is usually held in a tool-specific repository of some sort, similar in concept to the information provided in a CMP Entity bean's deployment descriptor, although typically much more complex and sophisticated. These O/R mapping products usually have sophisticated caching algorithms (for example, predictively loading related persistent data) that can radically speed up performance. On the other hand, one downside is that there is a learning curve to effectively configure and maintain the mapping data; sometimes professional assistance is needed. There are a number of O/R products around, including (in no particular order) TopLink, CocoBase, JavaBlend, JRelay, OJB, DBGen, JDX, and ObjectDriver.

Whether you can use any of these technologies may depend on the standards and constraints in your organization. Certainly, the vendors of OODBMS and O/R mapping products claim substantial decreases in the time taken to develop code.

As you will see shortly, one of the implicit objectives of JDO is to create a standard Java API for OODBMS and O/R mapping products. This may well cause these products to be adopted by a wider audience.

On the other hand, another consideration for you is the requirement for EJB container vendors to support CMP 2.0. You may find that some EJB vendors address this requirement simply by cross-licensing one of the more established O/R mapping tools. This is arguably the best of both worlds; you have access to a mature O/R mapping technology, but configured using an industry-standard EJB deployment descriptor.

JDBC

In the beginning, there was JDBC v1.0. Although initially introduced as an addendum to JDK 1.02, it was standardized as part of JDK 1.1 as the classes and interfaces that make up the `java.sql` package. It remains an integral part of J2SE 1.3. In an effort to unravel JDBC's family tree, Table 8.4 lists the versions of JDBC, J2SE (previously JDK), and J2EE.

TABLE 8.4 JDBC Versions

JDBC	*J2SE (JDK)*	*J2EE*	*Package*	*Significant Features/Notes*
JDBC 1.0	JDK 1.1	N/A	`java.sql`	`DriverManager`, `Connection`, `Statement`, `ResultSet` Note: Formalized in JDBC 1.2 Spec.
JDBC 2.1 Core API	J2SE 1.2	J2EE 1.2 J2EE 1.3	`java.sql`	Scrollable and updateable `ResultSets` Batch updates SQL1999 data types (`BLOB`, `CLOB`, `ARRAY`, Structured Type, `REF`) Mapping SQL UDTs to Java classes Direct storing of Java objects Note: When first introduced, was called the JDBC 2.0 Core API, not 2.1.
JDBC 2.0 Optional Package		J2EE 1.2 J2EE 1.3	`javax.sql`	`DataSource`, `ConnectionPoolDataSource`, `XADataSource` JNDI support Rowsets Note: When first introduced, was called the JDBC 2.0 Standard Extension API.
JDBC 3.0	J2SE 1.4	J2EE 1.4	java.sql javax.sql	Unifies JDBC 2.1 Core API and JDBC 2.0 Optional Package More thorough support for SQL1999 data types `DATALINK`/`URL` data type (external data) Savepoint support Retrieval of auto-generated keys Multiple open result sets Define relationship to Connector architecture Numerous other minor enhancements

The JDBC 2.1 Core API is included in J2SE 1.2 and also in J2EE 1.2 and J2EE 1.3 platforms. J2EE 1.2 and J2EE 1.3 also require the JDBC 2.0 Optional Package. If all these different versions aren't confusing enough, you may occasionally see documentation that refers to the "JDBC 2.0 API." This refers to the combination of (what is now called) the JDBC 2.1 Core API and JDBC 2.0 Optional Package. So, J2EE 1.2 and 1.3 effectively mandate the JDBC 2.0 API.

However, sanity is about to break out! At the time of writing, JDBC 3.0 was just coming out of draft, to be part of J2SE 1.4. This unifies both the Core API and the optional package. Given than J2SE 1.4 will include JDBC 3.0, it will also implicitly be part of J2EE 1.4.

One notable feature about JDBC 3.0 is that it consolidates support for SQL1999 advanced data types. Support for these was introduced in JDBC 2.1 Core API, when SQL1999 was still in draft and was called SQL3. Now that SQL1999 is a ratified standard, JDBC 3.0 has added some features that were, by necessity, previously omitted.

You can learn more about JDBC 3.0 (and download its specification) at `http://java.sun.com/products/jdbc/index.html`.

The advanced data types defined by SQL1999 are as follows:

- `BLOB`s are binary large objects. These can literally store anything, such as a JPEG image, a recording, a Java serialized object, and so on. `BLOB`s are transparently accessed using SQL `Locators`. The `BLOB` data isn't stored along with the rest of the row; instead, a pointer is held. While this marginally slows down access, it means that `BLOB`s can have very large size limits (2Gb or more).

  The JDBC `ResultSet.getBlob()` method and the `Blob` interface provide access to data stored in `BLOB` columns.

- `CLOB`s are character large objects, similar to `BLOB`s except that the data is treated as characters; as a result, conversion of data between locales is performed. `CLOB`s are also transparently accessed via SQL Locators.

  The JDBC `ResultSet.getClob()` method and the `Clob` interface provide access to data stored in `CLOB` columns.

- SQL structured types are a mechanism to allow user-defined data types to be defined. They are somewhat similar to a class in Java that has only public fields. The following SQL1999 command defines a structured type called `XY_POINT`:

  ```
  CREATE TYPE XY_POINT AS (X FLOAT, Y FLOAT) NOT FINAL
  ```

  SQL1999 allows columns to be defined of these types, and also allows tables to be defined consisting of instances of these types. Defining a table from a structure type is done using SQL syntax such as the following:

```
CREATE TABLE SCATTER_GRAPH OF XY_POINT (REF OID IS SYSTEM GENERATED);
```

The `OID` is an identifier to a row in this table. The JDBC `ResultSet.getObject()` method and the `Struct` interface provide access to data stored in structured type columns (not structured type tables). To simplify implementation, custom mappings can be defined to convert the data of the SQL structured type to a Java class. This is somewhat akin to implementing the `java.io.Externalizable` interface for the target Java class; `java.sql.SQLData` interface is the actual interface that must be implemented.

- `ARRAY`s provide the ability to store vector data. In the case study, the `Applicant` table defines two columns—`address1` and `address2`. Using SQL1999 types, these could instead be defined to be a single `ARRAY` column. Usually `ARRAY`s are accessed via SQL Locators, although it is possible for the data to be stored along with the rest of the data on the row.

  The JDBC `ResultSet.getArray()` method and the `Array` interface provide access to data stored in `ARRAY` columns.

- `REF`s are a persistent pointer to an instance of a SQL structured type, defined when a table is created from an SQL structured type. You might think of them as an SQL1999 equivalent to a foreign key.

  The JDBC `ResultSet.getRef()` method and the `Ref` interface provide access to the SQL `REF`. This reference can be used to access data held in a table defined to be of a structured type (such as `SCATTER_GRAPH`). This is the way that data stored in structured type tables can be accessed.

These data types may be unfamiliar to you; after all, many RDBMS do not support them yet. However, JDBC 3.0 includes support for these data types because the JDBC specification authors expect RDBMS support for them will have become widespread within the next five years or so.

That said, there is some overlap in intent between SQL1999 data types and the objectives of SQLj. If SQLj proves more popular (because it is rooted more on Java technology rather than SQL), it could be that Java's popularity may decelerate the adoption of SQL1999 data types. Only time will tell. So, high time to look at this next technology, SQLj.

SQLj

The SQLj initiative defines three ways to combine Java and SQL:

- SQLj Part 0 defines a mechanism for embedding SQL calls within Java code. The SQL is converted into JDBC calls by using a preprocessor.

- SQLj Part 1 inverts this, placing Java within SQL. It defines extensions to the SQL syntax to allow Java static methods to be called either as if they were SQL stored procedures or as user-defined functions within SQL statements.
- SQLj Part 2 also places Java within SQL, but here defining a mechanism for Java classes to be used to define SQL types. This allows table columns to be defined as a type of a Java class; a Java object instance can thus be stored directly in the column.

The SQLj Part 0 has already been adopted as an ANSI and ISO standard (ANSI ref: X3.135.10-1998, ISO/IEC ref: 9075-10:2000), "Information Technology—Database Languages—SQL—Part 10: Object Language Bindings (SQL/OLB)." If you see a reference to ANSI SQL Part 10, it just means SQLj Part 0.

The other parts have not yet been adopted by ANSI or ISO/IEC, but are being approved through NCITS, the (U.S.) National Committee for Information Technology Standards. Most of the NCITS standardization activities result in national (ANSI) or international (ISO/IEC) standards. SQLj Part 1 is also known as NCITS 331.1; SQLj Part 2 is NCITS 331.2.

The following sections look at each of these in turn.

SQLj Part 0

SQLj Part 0 is probably the most straightforward for database vendors to implement, because it an entirely a client-side technology. The developer, rather than writing a regular Java class whose file has a `.java` suffix, writes a hybrid class instead that has embedded SQL commands within it. Such a class resides in a file with a `.sqlj` suffix. A preprocessor (SQLj calls it a *translator*) converts the embedded SQL commands into JDBC equivalents, resulting in a `.java` file that can be compiled in the usual way. The `sqlj` translator also creates a set of serialized `.ser` profile files that hold information about the embedded SQL.

On a historical note, EJB 1.0 used `.ser` files to hold deployment descriptor information (as you know, XML files are now used instead). SQLj Part 0 was initially specified at around the same time as EJB 1.0 (1998 or so), so it is interesting to speculate if it were being specified now whether it would use XML files to hold this supplementary information.

After the generated `.java` and `.ser` files have been created, they can be compiled and packaged up into an EJB JAR file in the usual way. The `.ser` profile files simply need to be part of the `ejb-jar`. This overall process is shown in Figure 8.11.

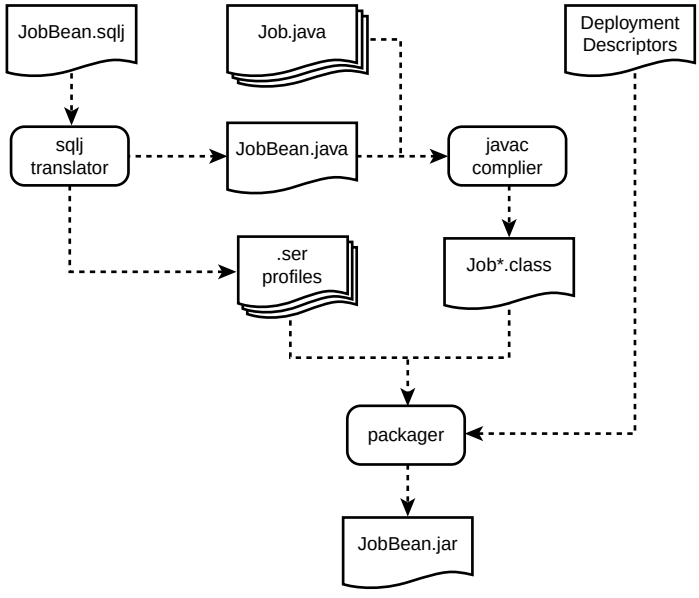


FIGURE 8.11
SQLj Part 0 uses a translator to convert SQLj commands into JDBC.

The `sqlj` translator can optionally also check the embedded SQL against a target database to make sure that it is syntactically correct (that all tables, columns, and so on that are referenced do exist).

Listing 8.5 shows the BMP version of the `ejbLoad()` method for the `Job` Entity bean (from Day 7), written using JDBC.

LISTING 8.5 JDBC Version of `ejbLoad()` for BMP `Job` Bean

```
 1: public void ejbLoad(){
 2:     JobPK key= (JobPK)ctx.getPrimaryKey();
 3:     Connection con = null;
 4:     PreparedStatement stmt = null;
 5:     ResultSet rs = null;
 6:     try {
 7:         con = dataSource.getConnection();
 8:         stmt = con.prepareStatement(
 9:         "SELECT description,location
           ➥FROM Job
           ➥WHERE ref = ?
           ➥AND customer = ?");
10:
11:         stmt.setString(1, key.getRef());
12:         stmt.setString(2, key.getCustomer());
13:         rs = stmt.executeQuery();
14:
```

LISTING 8.5 Continued

```
15:         if (!rs.next()) {
16:            error("No data found in ejbLoad for "+key,null);
17:         }
18:         this.ref = key.getRef();
19:         this.customer = key.getCustomer();
20:         this.customerObj =
            ➥customerHome.findByPrimaryKey(this.customer); // derived
21:         this.description = rs.getString(1);
22:         String locationName = rs.getString(2);
23:         this.location =
            ➥(locationName != null)?
            ➥locationHome.findByPrimaryKey(locationName):null;
24:
25:         // load skills
26:         stmt = con.prepareStatement(
27:         "SELECT job, customer, skill
            ➥FROM JobSkill
            ➥WHERE job = ?
            ➥AND customer = ?
            ➥ORDER BY skill");
28:
29:         stmt.setString(1, ref);
30:         stmt.setString(2, customerObj.getLogin());
31:         rs = stmt.executeQuery();
32:
33:         List skillNameList = new ArrayList();
34:         while (rs.next()) {
35:             skillNameList.add(rs.getString(3));
36:         }
37:
38:         this.skills = skillHome.lookup(skillNameList);
39:     }
40:     catch (SQLException e) {
41:         error("Error in ejbLoad for "+key,e);
42:     }
43:     catch (FinderException e) {
44:         error("Error in ejbLoad (invalid customer or location) for "+
                  ➥key,e);
45:     }
46:     finally {
47:         closeConnection(con, stmt, rs);
48:     }
49: }
```

Listing 8.6 shows the same method, implemented using SQLj embedded SQL syntax.

LISTING 8.6 SQLj Version of `ejbLoad()` for BMP `Job` Bean

```
 1: // assuming
 2: // import sqlj.runtime.*;
 3:
 4: public void ejbLoad(){
 5:     JobPK key= (JobPK)ctx.getPrimaryKey();
 6:
 7:     #sql context CustomConnectionContext;
 8:     CustomConnectionContext conCtx;
 9:     try {
10:         con = dataSource.getConnection();
11:         conCtx = new CustomConnectionContext(con);
12:         #sql [conCtx]
13:              { SELECT description,location
14:                INTO :this.description, :this.locationName
15:                FROM Job WHERE ref = :(key.getRef())
16:                AND customer = :(key.getCustomer())   };
17:
18:          this.ref = key.getRef();
19:          this.customer = key.getCustomer();
20:          this.customerObj =
             ➥customerHome.findByPrimaryKey(this.customer); // derived
21:          this.location =
             ➥(locationName != null)?
             ➥locationHome.findByPrimaryKey(locationName):null;
22:
23:         // load skills
24:         #sql iterator SkillIterator
            ➥(String job, String customer, String skill);
25:         SkillIterator skillIter =
26:             #sql [conCtx]
27:                 { SELECT job, customer, skill
28:                   FROM JobSkill
29:                   WHERE job = :this.ref
30:                   AND   customer = :(customerObj.getLogin())
31:                   ORDER BY skill };
32:
33:         List skillNameList = new ArrayList();
34:         while (skillIter.next()) {
35:             skillNameList.add(skillIter.skill());
36:         }
37:         skillIter.close();
38:
39:         this.skills = skillHome.lookup(skillNameList);
40:     }
41:     catch (SQLException e) {
42:         error("Error in ejbLoad for "+key,e);
43:     }
44:     catch (FinderException e) {
```

LISTING 8.6 Continued

```
45:         error("Error in ejbLoad (invalid customer or location) for "+
                ➥key,e);
46:     }
47:     finally {
48:         conCtx.close(ConnectionContext.KEEP_CONNECTION);
49:         closeConnection(con, null, null);
50:     }
51: }
```

You can see that SQLj binds host variables (that is, Java instance or local variables) or return values of expressions to the SQL statement using a `:` prefix. You can see this in the first SQL statement (loading from the `Job` table) where two function expressions are used in the `WHERE` clause. Host variables can also be written to through the `INTO` clause. Again, in the first SQL statement, the `INTO` clause is used to populate the `location` and `description` host variables. Contrast the use of the `INTO` clause with the JDBC equivalent that has to iterate over a `java.sql.ResultSet`.

The second SQL statement is a query against the `JobSkill` table, and multiple rows are expected this time. A typesafe iterator, `SkillIterator`, is defined and is used to traverse the returned rows.

Connection information can be specified in SQLj in a variety of ways. The normal approach is to embed the connection information into the aforementioned `.ser` profile files, replacing the J2SE `java.sql.DriverManager` approach. Obviously, this isn't appropriate in a J2EE environment, because only logical connections should be acquired via `javax.sql.DataSource`.

One solution is to use an explicit *connection context* to define the connection that the SQL will be executed under. In Listing 8.6, the line

```
#sql context CustomConnectionContext;
```

defines `CustomConnectionContext` as a user-defined class, extending `sqlj.runtime.ConnectionContext` and with a constructor accepting a `java.sql.Connection`. The line

```
CustomConnectionContext conCtx;
```

defines a reference `conCtx` of this class, and

```
con = dataSource.getConnection();
conCtx = new CustomConnectionContext(con);
```

instantiates an object of this class. The `conCtx` is used within the subsequent `#sql` calls.

You may find that your vendor's SQLj implementation offers the ability to make a `CustomConnectionContext` the default context. Part of the SQLj Part 0 specification includes the concept of customizing profiles to a specific runtime environment, similar to the notion of adding the auxiliary deployment descriptor for EJBs. Your customizer may allow you to indicate that the default `ConnectionContext` is one that is "J2EE-aware." This would avoid the need for any of the connection context code given in Listing 8.6.

SQLj Part 1

Whereas SQLj Part 0 embedded SQL within Java, the SQLj Part 1 specification inverts this and puts the Java within SQL. More properly, it allows Java static methods to be called as if they were SQL stored procedures. It also allows Java static methods to act as user-defined functions called from within SQL statements (such as `SELECT` or `UPDATE`).

What this means is that, whereas SQLj Part 0 addresses Java and SQL for client-side code, SQLj Part 1 puts Java and SQL together in the server. Put another way, SQLj Part 1 can only be supported by RDBMS vendors that provide the capability to install this functionality.

There are two main mechanisms by which RDBMS vendors can support SQLj Part 1. Many vendors have long provided the ability to hook user-defined functions into the RDBMS, where those functions are written using C or some other 3GL. Some such vendors offer SQLj Part 1 support by converting the Java static methods into corresponding C code and then uploading.

Alternatively, the RDBMS vendor can actually implement a JVM that runs within the RDBMS itself. When the Java static method is called, the SQL "engine" within the RDBMS passes over control to the JVM to evaluate the result.

As an example, consider the `Applicant` table from the case study. This table records each applicant's address in the `address1` and `address2` columns. For the sake of illustration, imagine that it also has the `state`, `zip`, and `country` columns.

If the Job agency users want to do a mailshot, they will need to print some labels. The labels need to combine the information of the `name`, `address1`, `address2`, `state`, `zip`, and `country` columns, with a newline between each, also dealing with the case where some of these columns are null. Listing 8.7 defines a utility class that will format the label string appropriately.

LISTING 8.7 Static Methods Can Be Invoked as SQL UDFs

```
 1: public class Utils {
 2:     public static String labelString(
 3:             String name, String address1, String address2,
 4              String state, String zip, String country) {
 5:         StringBuffer label = new StringBuffer(name);
 6:         if (address1 != null) {
 7:             label.append("\n"); label.append(address1);
 8:         }
 9:         if (address2 != null) {
10:             label.append("\n"); label.append(address2);
11:         }
12:         if (state    != null) {
13:             label.append("\n"); label.append(state   );
14:         }
15:         if (zip      != null) {
16:             label.append("\n"); label.append(zip     );
17:         }
18:         if (country  != null) {
19:             label.append("\n"); label.append(country );
20:         }
21:         return label.toString();
22:     }
23: }
```

This class would be compiled as usual and bundled up into a JAR file, say `utils.jar`. SQLj defines the following mechanism for installing classes as JAR files:

```
sqlj.install_jar ('file:utils.jar', utils_jar' )
```

This gives the name `utils_jar` to the JAR file `utils.jar`. Each RDBMS vendor will provide a tool that implements this mechanism.

With the code installed, the following defines the user-defined function `label_for` to be the `Utils.labelString()` method:

```
create function label_for(
name varchar, address1 varchar, address2 varchar,
    state char(2), zip char(9) country varchar)
returns varchar
language java
parameter style java
external name 'utils_jar:Utils.labelString';
```

Finally, the following SQL can be executed:

```
SELECT label_for(name, address1, address2, state, zip, country)
FROM Applicant
```

Using a function alias means that access to this function can be granted or revoked as needed by using the SQL `grant execute` command. Additionally, SQLj specifies that security can be associated with the actual JAR file using `grant usage on utils_jar to some_user`.

Some RDBMS vendors have relaxed the requirement to define function aliases and simply allow the Java static method to be called directly:

```
SELECT Utils.labelString(name, address1, address2,
                         state, zip, country) AS label
FROM Applicant
```

One of the major RDBMS vendors (Sybase) allows Java UDFs to read/write `BLOB` (`image`) and `CLOB` (`text`) columns as `java.io.InputStream` and `java.io.Reader`s, respectively. So, given a table of

```
create table letter
( letter_id int,
  letter_text text )
```

and a static method such as

```
public class TextUtils {
    public static int length(java.io.Reader reader) {
        int size = 0, read;
        char[] buf = new char[1024];
        while( (read = reader.read(buf)) != -1) {
            size += read;
        }
        return size;
    }
}
```

then SQL such as the following can be submitted:

```
SELECT letter_id, TextUtils.length(text)
FROM   letters
```

Clearly, this opens up a lot of possibilities. Rather than just counting the number of characters in the `CLOB` column, the Java method could search, apply regular expressions, run AWK scripts, or parse XML.

Indeed, the RDBMS vendor has developed an XML query engine implemented in precisely this fashion. As a result, developers can store XML files as `CLOB`s and process them within the RDBMS server itself.

SQLj Part 1 also allows stored procedures to be defined as an alias for Java static methods. So, a Java client making a JDBC call such as the following

```
CallableStatement stmt = con.prepareCall("{ call some_such_procedure ?, ?}";
stmt.execute();
```

is actually invoking a Java static method aliased to `some_such_procedure`. This is almost like performing a Java RMI call, except over a database network connection.

Such "Java stored procedures" can (somewhat arbitrarily) be split into two types—those that have embedded SQL calls implemented using JDBC or SQLj and those that do not.

Java stored procedures that do *not* have embedded SQL often perform some sort of complex computation that is not easily expressed in SQL. As a somewhat contrived example, Listing 8.8 shows a reworked version of the `Utils.labelString` method that allows it to be called as a stored procedure.

LISTING 8.8 Static Methods Can Be Invoked as SQL Stored Procedures

```
 1: public class Utils {
 2:     public static String labelString(
 3:             String name, String address1, String address2,
 4:             String state, String zip, String country,
 5:             String[] returnLabel) {
 6:         StringBuffer label = new StringBuffer(name);
 7:         if (address1 != null) {
 8:             label.append("\n"); label.append(address1);
 9:         }
10:         if (address2 != null) {
11:             label.append("\n"); label.append(address2);
12:         }
13:         if (state    != null) {
14:             label.append("\n"); label.append(state   );
15:         }
16:         if (zip      != null) {
17:             label.append("\n"); label.append(zip     );
18:         }
19:         if (country  != null) {
20:             label.append("\n"); label.append(country );
21:         }
22:         returnLabel[0] = label.toString();
23:     }
24:     //  code omitted
25: }
```

The stored procedure definition for this method is as follows:

```
create procedure label_it
 (in name varchar, in address1 varchar, in address2 varchar,
  in state char(2), in zip char(9), in country varchar,
out returnLabel varchar)
language java
parameter style java
external name 'utils_jar;Utils.labelString';
```

The following code will work to invoke this stored procedure from JDBC:

```
CallableStatement stmt = con.prepareCall (
        ➥"{call label_it ?, ?, ?, ?, ?, ?, ? }");
stmt.setParameter(1, someName);
stmt.setParameter(2, someAddress1);
stmt.setParameter(3, someAddress2);
stmt.setParameter(4, someState);
stmt.setParameter(5, someZip);
stmt.setParameter(6, someCountry);
String theReturnedLabel;
stmt.registerOutParameter(7, Types.VARCHAR );
stmt.setParameter(7, theReturnedLabel);
stmt.execute ();
```

You may have noticed in Listing 8.8 the peculiar fact that the output parameter, `returnLabel`, is defined to be an array of `String`s, rather than just a `String`. This is no mistake; the method needs to be able to change the value of the parameter. The array passed in has precisely one element, the initial value of the `returnLabel`, and can be changed.

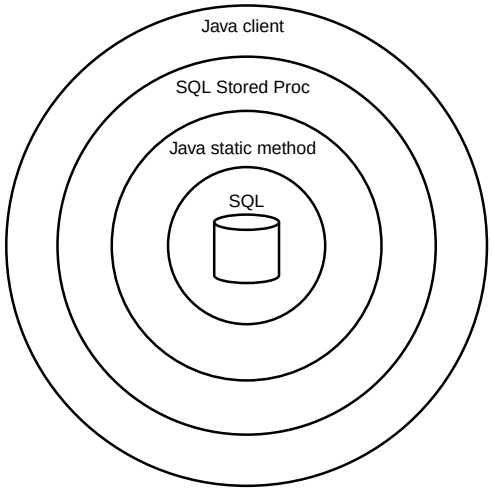
If you are a C or C++ programmer, you might recognize this as the Java equivalent of "pointer to a pointer."

You can probably think of some less contrived examples yourself.

The other type of Java stored procedure is that which *does* have embedded SQL. Here, you get the peculiar situation of a Java client calling SQL, which maps to Java calling SQL. This brings to mind concentric rings of Java and SQL, as shown in Figure 8.12.

FIGURE 8.12

SQL embedded within Java, embedded within SQL, embedded within Java…



The outermost ring is a Java client (Session bean, CMP Entity bean, servlet, and so on). The inner three rings are server-side, running within the RDBMS. The Java client invokes an SQL stored procedure, in turn aliased to a Java static method that then makes SQL calls.

For the Java stored procedure to query the database, it must have a JDBC connection. SQLj defines the URL string of `jdbc:default:connection` to be used as the parameter to `java.sql.DriverManager.getConnection()`.

Not only can Java static methods query the database, they can also return the results. This is done by allowing `ResultSets` to be returned as parameters, in a manner similar to the `returnLabel` that you saw earlier.

This second style of Java stored procedure is really just the re-location of a client-side application. In other words, with appropriate modifications to the mechanism to obtain the connection, the code could equally run in a regular Java client. Put another way, there is no difference in the amount of JDBC or embedded SQL code that must be written. As a result, it can seem that as a technology, this is a curiosity—nothing more. However, that would be overlooking the important point that running Java client-side code within the RDBMS itself means that there is no network traffic. Many batch processing jobs involve downloading large chunks of data, performing complex processing, and then uploading the results. Using Java stored procedures, all of this work can be done without incurring any costly network traffic at all.

By making Java static methods executable within the RDBMS, SQLj Part 1 offers some quite powerful functionality. It is a wonderful demonstration of Java's oft-quoted "write once, run anywhere" ability.

SQLj Part 2

The last part of SQLj is possibly the most interesting of all. SQLj Part 2 allows Java classes to be defined as SQL abstract data types.

You will probably recall from the earlier discussion on SQL1999 advanced data types that there is overlap between SQL1999 and SQLj Part 2. The most obvious overlap is that Java classes can be used instead of SQL structured types.

Listing 8.9 shows a class that represents all the states of a `Job` bean. This includes also the (names of the) job's skills and the (name of the) job's location.

LISTING 8.9 The `JobData` Class Represents All the State of a `Job` Bean

```
 1: package data;
 2: public class JobData implements java.io.Serializable {
 3:     public final static long serialVersionUID = 1;
 4:     private String ref, customer, description, locationName;
 5:     private String[] skillNames;
 6:     public JobData(String ref, String customer,
                    ➥String description,
                    ➥String locationName,
                    ➥String[] skillNames) {
 7:         this.ref = ref;
 8:         this.customer = customer;
 9:         this.description = description;
10:         this.locationName = locationName;
11:         if (skillNames = null) { skillNames = new String[0]; }
12:         this.skillNames = new String[skillNames.length];
13:         System.arraycopy(skillNames, 0,
            ➥this.skillNames, 0, skillNames.length);
14:     }
15:     public String getRef() { return ref; }
16:     public String getCustomer() { return customer; }
17:     public String getDescription() { return description; }
18:     public String getLocationName() { return locationName; }
19:     public String[] getSkillNames() {
20:         String[] skillNames = new String[this.skillNames.length];
21:         System.arraycopy(this.skillNames, 0,
            ➥skillNames, 0, skillNames.length);
22:         return skillNames;
23:     }
24:     public int getNumberOfSkills() { return this.skillNames.length; }
25: }
```

The class implements `java.io.Serializable` and defines `serialVersionUID` to provide forward compatibility with future versions of the class.

This class can be bundled into a JAR and installed into the RDBMS, just as for SQLj Part 1. This time though, a user-defined type can be created from the Java class, using an extension of the SQL1999 `CREATE TYPE` syntax:

```
CREATE TYPE JobType EXTERNAL NAME 'data.JobData' LANGUAGE java;
```

This example is incomplete because SQLj Part 2 is still in development. Some RDBMS vendors that have supported SQLj Part 2 have used alternative approaches. For example, the Cloudscape RDBMS uses the following syntax:

```
CREATE CLASS ALIAS JobType FOR data.JobData;
```

Even then, this is an optional step in Cloudscape, as it is for the Sybase RDBMS. That is, the fully-qualified Java classname can just be used directly as a column type. Consequently, the following could be used to define the `Job` table:

```
CREATE TABLE Job
(
   job data.JobData
)
```

Each row in the table will hold a serialized instance of the `data.JobData` class. Performance-wise, this would be prohibitively expensive to access unless the RDBMS vendor has implemented a JVM within the RDBMS. If they have done this though, the cost of deserializing and serializing the objects is not expensive. The RDBMS has suddenly become an ORDBMS (object/relational database).

When the data in the table is actually a Java object, suddenly SQL queries become a lot more exciting:

```
SELECT job>>getCustomer(), avg( job>>getNumberOfSkills() )
FROM Job
GROUP BY job>>getCustomer()
```

This query will find the average number of skills needed for all of the jobs placed by a customer.

In SQLj Part 2, the >> operator replaces the conventional . notation.

The following query also works:

```
SELECT job
FROM Job
WHERE job>>getNumberOfSkills() > 2
```

The calling client's JDBC is simplicity itself:

```
ResultSet rs = stmt.execute(
➥"SELECT job FROM Job WHERE job>>getNumberOfSkills() > 2");
while (rs.next()) {
    JobData job = (JobData)rs.getObject(1);
}
```

Although this is slightly off-topic, it is worth noting that SQLj Part 2 transparently supports super- and sub-types. One could define a subclass of `JobData` and store instances of it within the `Job` table. Moreover, any overridden methods would be called polymorphically. This feature alone could radically simplify many database schemas.

Turning back to the SQL1999 data types, you can probably see that Java classes can be used in lieu of a `BLOB`, `CLOB`, or `ARRAY`, as well as be replaced SQL structured types (for columns). The following is another definition of the `Job` table, this time showing `skills` as a vector attribute:

```
CREATE TABLE Job
(
    customer VARCHAR,
    ref VARCHAR,
    description VARCHAR,
    location VARCHAR,
    skills java.util.List
)
```

One could then imagine the following query:

```
SELECT customer, ref
FROM Job
WHERE skills>>contains("Cigar trimmer")
```

The Cloudscape RDBMS goes even further, providing implicit mapping of regular SQL types (such as `VARCHAR`) to their Java equivalents (such as `java.lang.String`). Hence, the following query would be valid within Cloudscape:

```
SELECT customer>>substring(0,2)>>concat(ref)>>length()
FROM Job
WHERE customer>>startsWith("XYZ")
```

Of the SQL1999 data types, only the `REF` data type has no direct equivalent in SQLj. Indeed, if you go back to the first alternate definition of the `Job` table (defined in terms of the `data.JobData` class), you can see that the `job` object has the *name* of the location, not a reference to the location. If one were to put this in EJB terms, one might say that `JobData` holds the primary key to the job's location, not a reference to `LocationLocal`.

There are some other issues that you should be aware of:

- First is that, given the actual data is held as a Java serialized object, non-Java database clients will not be able to read that data.

  Possible fixes for this issue are to

  - Write code to read the (well-defined) Java serialization stream
  - Implement `java.io.Externalizable` and provide custom methods to write the state in some other format (XML, perhaps?)

- Second, the RDBMS' query optimizer cannot index on the return value of method calls. For example, looking back at the `WHERE` clause of one of the previous examples (`WHERE job>>getNumberOfSkills() > 2 `), this can only be performed by deserializing every object and invoking the `getNumberOfSkills()` method.

  One possible fix is to redundantly store the required information in a regular column, and maintain it using triggers. The query then uses this derived column.

True OODBMS do allow indexes to be defined on the return values of read-only methods.

- Last, this is new technology, so it will need to mature before organizations are ready to trust their valuable data to it.

How does SQLj (parts 1 and 2) fit with EJB and the J2EE platform? Well, the support in SQLj Part 2 certainly makes implementing BMP Entity beans pretty straightforward, provided care is taken with relationships:

- If a BMP Entity bean is the parent of a one-to-many composite association, there is nothing to prevent that bean from simply persisting the dependent child data and dispensing with the child table in the RDBMS.
- Many-to-one associations, where the bean is a child, are more complex. In general, only the foreign key value should be persisted in the bean. This does incur extra cost for looking up the actual parent data. (There is an exception to this, in the special case where the parent is immutable. Then, the parent's data can safely be stored in the bean itself.)
- Many-to-many associations can be treated like one-to-many associations, but only if the data that is held is the foreign key value to the other table, and also provided that the many-to-many link does not need to be traversed in both ways.

One of the previous examples stored the job skills as a `java.util.List` of `Strings`, which seems elegant, but to identify which jobs require the "Cigar trimmer" skill necessitates instantiating every job instance; in other words, always coming at the `Job`/`Skill` association from the `Job` side.

Even with these provisos, one could imagine that it would be relatively easy (given a modern IDE or UML modeling tool) to automatically generate SQLj implementation code for BMP Entity beans.

On the other hand, the ability to deploy Java code that runs within the RDBMS itself (using either SQLj Part 1 or Part 2) somewhat muddies the architectural water. You are by now intimately familiar with the n-tier architecture model. But, if Java can run in the RDBMS, why need do you have a middle tier at all? This is a question that you are probably best advised to answer yourself. You may take the view that as long as you know which *logical* layer your Java code belongs to, it may not matter which physical tier (Web server, EJB application server, Java-enabled RDBMS server) you choose to deploy it to.

JDO

The last persistence technology that you will be looking at, Java Data Objects (JDO), aims high and it aims low. That is, it is intended both for use within embedded devices (J2ME) but also for use within J2EE environments. In a J2EE environment, JDO can either supplement or supplant Entity EJBs; if supplementing Entity EJBs, the application of JDO may be hidden if CMP is being employed (that is, the EJB vendor uses JDO but the bean provider is unaware of this), or it can be used by the bean provider directly if BMP is in use.

JDO is the most recent of the Java persistent technologies and, at the time of writing, its specification was still in draft with only an incomplete reference implementation. Moreover, as a technology it seems to be running in parallel with the various Java platform editions (J2ME, J2SE, and J2EE), and there appear to be no clear indications within which platform it will eventually reside. Nevertheless, JDO has raised some significant interest as an API, especially for vendors of OODBMS and O/R mapping tools. Indeed, the JDO expert group includes representatives from Versant, Poet, Object Design, and Gemstone, among others. Because JDO completely hides the details of the data store internals, it is also suitable to access ERP systems; some ERP vendors (such as SAP) have committed to adopting it.

One way to think of JDO is as a vendor-neutral Java API to OODBMS and O/R mapping tools, just as JDBC provides a vendor-neutral Java API to RDBMS.

The overriding objective of JDO is to provide persistence transparency. In other words, the objects that are to be persisted—so-called *persistent-capable* objects—do not need to include *any* logic to make them persistent. Contrast this with EJB Entity beans that need to implement `ejbLoad()` and `ejbStore()`, or with EJB Session beans (and for that matter servlets) that require reams of JDBC or SQLj calls. The only classes that need to use the JDO interfaces and classes are those that manage the lifecycle of persistent objects.

As an example of transparent persistence, consider that persistent-capable objects can also have references to other objects. Assuming that the field that holds this reference has been marked as being persistent, these referenced classes will then also be made persistent. This is sometimes called "persistence by reachability." Again, contrast this to EJB, where a relationship between beans required complex configuration in the EJB deployment descriptor, moreover being constrained to relate through the EJB's local interface.

It is perhaps a little misleading to claim that JDO does not need relationships between classes to be defined. Rather, it is not within the scope of JDO to be concerned about this. A JDO implementation provided by an OODBMS vendor will work in a different way than one provided by an O/R mapping tool vendor. Any mapping or other configuration information that might need to be done is performed entirely with vendor-specific tools.

Earlier, JDO was compared to JDBC. JDBC provides a standard API, but leaves RDBMS vendors at liberty to implement their own data store and network protocols. Equally, JDO provides just an API and does not get involved in the internals. This makes JDO applications portable across JDO implementations, but obviously requires any vendor-specific configuration to be re-applied.

JDO Concepts

JDO's approach to object persistence revolves around the concept of a cache. This cache belongs to a client, rather than the server; the objects in the cache are not shared among all clients. In a J2EE environment, a stateful Session bean would usually be a client; each active Session bean would have its own cache. At any given time, the cache that holds the objects is associated with at most one connection and at most one transaction.

Over time, that cache can be used with different transactions, and, for that matter, with different connections.

The JDO interface that controls the client's cache is `javax.jdo.PersistenceManager`. In a J2EE environment, an instance of this interface is obtained from a `javax.jdo.PersistenceManagerFactory` that, in turn, is obtained via JNDI. The JDO specification describes how the J2EE Connector architecture is used to actually configure a `PersistenceManagerFactory` into JNDI; you'll be learning something about this on Day 19, "Integrating with External Resources." All you need to appreciate for now is that a JDO vendor (an OODBMS vendor, O/R mapping tool vendor, or ERP vendor) will have implemented the appropriate J2EE Connector interfaces such that a `PersistenceManagerFactory` will be available for you to look up.

JDO also integrates with XA, meaning that distributed transactions across multiple JDO implementations, and indeed RDBMS data stores, are supported.

JDO defines two main ways in which the cache can be used. All JDO implementations must support so-called *data store transactions* and can optionally support *optimistic transactions*:

- When a persistent-capable object is in a cache through a data store transaction, it effectively prevents any other user from holding this object in his or her cache.

  It is almost as if the object is "checked out" to the user's cache, a little like checking out code from a source code control system. It remains there for that user's exclusive use (to be read or modified) until he or she indicates that the transaction is complete. Any other user who wants to use the object must wait until the original user's transaction has completed.

  You may perhaps recognize this approach; it is often called *pessimistic locking*.

- Although not mandatory within the JDO specification, most JDO implementations suitable for use within a J2EE environment will also support optimistic locking.

  Here, it is possible for a persistent-capable object to reside in two different users' caches at the same time. As long as each only reads the data from the object, there are no issues (hence, "optimistic").

  If one user modifies the data and commits their optimistic transaction, the new state of the object will be transparently written back to the data store. The other user does not necessarily know about this (though he or she can request to refresh the object instance if needed).

> Of course, issues do arise when both users modify the data. In this case, the first user to commit his or her transaction will succeed. When the second user attempts to commit his or her transaction, the `PersistenceManager` will throw a `javax.jdo.JDOUserException`, detailing that the state of the object has been changed by some other user since it was first instantiated in the cache.

From the data store's perspective, the optimistic transaction approach actually involves two data store transactions. The first is short-lived, lasting long enough just to read the data from the data store. The second will happen some time later (perhaps seconds, minutes, or longer) and again will be short-lived. During this second data store transaction, the persistent data will be updated with the modified data taken from the committing user's cache.

The JDO specification also defines the notion of JDO identity. There are three different ways for a JDO vendor to implement JDO identity, depending on the underlying technology:

- For JDO implementations based on an O/R mapping tool, JDO identity basically corresponds to the primary key as defined in the RDBMS. The JDO specification terms this *application identity* or *primary key identity*.
- For JDO implementations based on an OODBMS, JDO identity corresponds to the object ID as assigned by the OODBMS itself. The JDO specification calls this *data store identity*.
- The final JDO identity type is perhaps likely to be least often used; it is simply called *non-data store identity*. This relates to objects unique within the JVM, but that are only ever written to (not read from) a data store (entries in a log file, for example).

The term *application (primary key) identity* is used for RDBMS-based identity because it is effectively the application itself that defines the semantics of the primary key. This has echoes in EJB where you, as the bean developer, are required to implement a primary key class or identify the primary key field. The RDBMS data store is expected to enforce the notion of identity through the use of unique indexes.

JDO identity is not the same as Java object identity, because in a single JVM, there could be many active `PersistenceManagers`; if optimistic transactions are used, multiple Java objects could be instantiated all with the same JDO identity. However, JDO identity does have some correspondence to Java's notion of equality (that is, where two objects are considered the same if `equals()` returns `true`). In particular, JDO requires that persistence-capable objects implement `equals()` in such a way that it returns `true` if and only if the JDO identity is the same. Unsurprisingly, this is exactly the requirement that EJB imposes on primary keys.

> JDO is very clear about identity, recognizing that more than one object instance may be instantiated for a single instance in the data store. In contrast, EJB is quite casual in this regard. Indeed, the EJB specification explicitly states that EJB containers have latitude to, for example, implement precisely one Entity bean per instance in the data store (option A, section 10.5.9) or to have multiple instances (option C).
>
> Such latitude is surprising. Option A effectively means that Entity beans reside in a server-side cache. One consequence of this is that deadlocks must be handled in the EJB container. Meanwhile, option C effectively means that Entity beans reside in a client-side cache; any deadlocks arising are handled in the data store.

For any given persistence-capable object, only one of these different types of JDO identity applies. The JDO deployment descriptor (described briefly later) identifies the type of JDO identity in use.

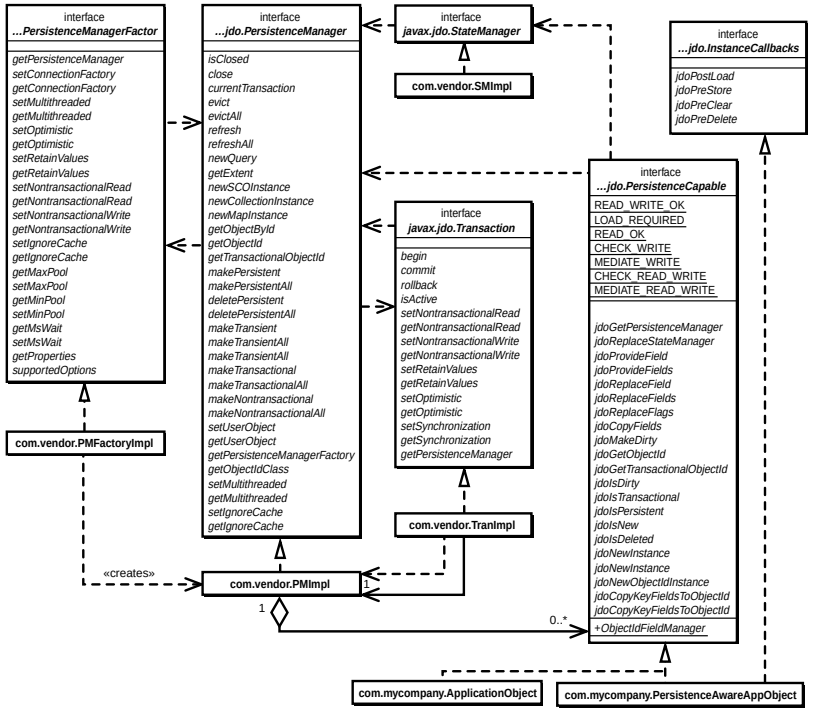
`javax.jdo` Classes and Interfaces

The JDO API is defined by classes and interfaces in the `javax.jdo` package. Some of the interfaces are intended to be implemented by the JDO vendor and some by the application developer (but more on this in a moment). Figure 8.13 shows the main classes of the `javax.jdo` package.

The `javax.jdo.PersistenceManagerFactory`, `javax.jdo.PersistenceManager`, `javax.jdo.StateManager`, and `javax.jdo.Transaction` interfaces are all implemented by the JDO vendor. The `PersistenceManagerFactory` has already been discussed. The `PersistenceManager` is the most important interface, because it provides the methods to control the lifecycle of the persistence-capable objects. Relating this back to EJB, you might think of it as combining the functions of the EJB container and of an EJB's home interface. The `PersistenceManager` also provides access to the current `javax.jdo.Transaction`. This allows the application developer to demarcate the transaction boundaries. In a J2EE environment, a CMTD EJB does not need to call the methods of this interface because the EJB container will do this work. If a BMTD Session bean is being developed to use JDO, the bean developer can use either the `javax.jdo.Transaction` or the `javax.transaction.UserTransaction` interface. The former is usually to be preferred however, because this allows a single `PersistenceManager` to be used across multiple transactions. If a `UserTransaction` is used, a `PersistenceManager` must be acquired for each and every transaction. Finally, the `StateManager` is used internally by the JDO implementation to keep track of changes to persistence-capable objects.

FIGURE 8.13

The significant classes and interfaces of the `javax.jdo` package.



Of the two remaining interfaces shown in Figure 8.13, the `javax.jdo.PersistenceCapable` interface is implemented by the application developer. Hang on though! Isn't JDO meant to support transparent persistence? Doesn't that mean that there should be no requirement for the application classes to implement any sort of interface at all? Well, yes…and no. It is true that the persistence-capable classes developed by the application developer neither need to call the classes in `javax.jdo`, nor do they need to implement any of the interfaces in `javax.jdo`. However, before a persistent-capable class can be deployed into a JDO Implementation, it must be "enhanced." An enhancer is a tool provided by the JDO vendor that manipulates the byte code of the compiled application classes. The resulting enhanced byte code represents a version of the application class that *does* implement the `PersistenceCapable` interface.

This whole process possibly sounds somewhat peculiar, but compare it to the approach used in EJB for CMP Entity beans. As you recall, the bean developer creates an abstract class with abstract getter and setter methods for each of the `cmp-field`s. The EJB container vendor's deployment tools then generate an implementation for those methods and create the database access code. The enhancement process effectively does the equivalent of the first of these two tasks and also imbues the application class with a reference to a `StateManager` instance. The `StateManager` itself takes on the job of database access and persistence.

The final interface, shown in Figure 8.13, is the `InstanceCallbacks` interface. The application developer can choose to implement this or not. If implemented, the methods give the application class visibility as to the transaction boundaries. You may have spotted that this is pretty similar to EJB's optional `SessionSynchronization` interface.

Time for some code! Listing 8.10 shows the basic steps to create a new persistence-capable `Customer`.

LISTING 8.10 Using JDO to Create a New `Customer`

```
 1: // import javax.jdo.*;
 2: // import javax.naming.*;
 3:
 4: Context ctx = new InitialContext();
 5: PersistenceManagerFactory pmf = (PersistenceManagerFactory)
 6:     ctx.lookup("java:comp/env/jdo/SomePersistenceManagerFactory";
 7: PersistenceManager pm = pmf.getPersistenceManager();
 8:
 9: Transaction txn = pm.currentTransaction();
10: txn.begin();
11: Customer customer = new Customer(login, address1, address2, email, name);
12: pm.makePersistent(customer);
13: txn.commit();
```

That's honestly all there is to it! `Customer` is just a regular Java class that has been run through the JDO Enhancer.

Queries

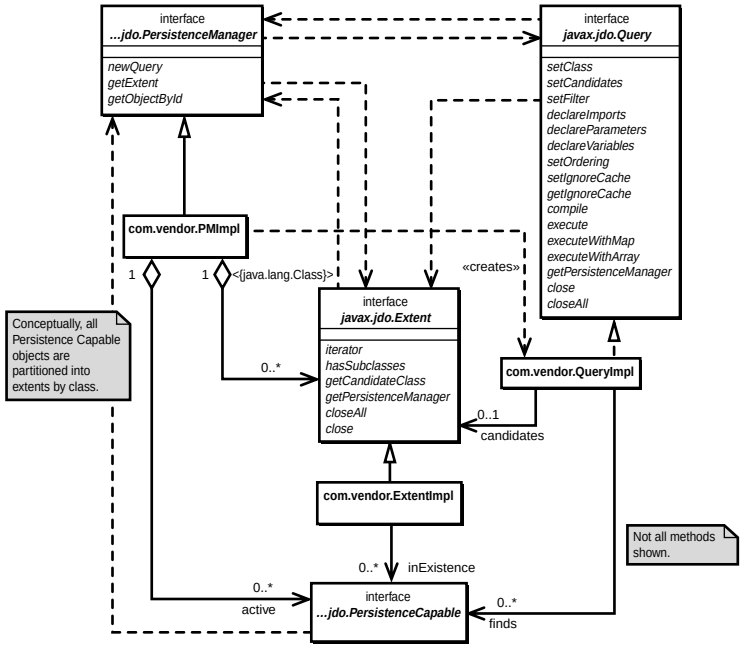
As well as being able to create new objects, JDO also provides the facility to find existing objects. This is not surprising; after all, the home interface offers the finder methods as well as the create methods in EJB.

Rather than defining a declarative language, such as SQL or EJBQL, JDO uses a Java API approach. Figure 8.14 shows the classes and interfaces of `javax.jdo` that allow queries to be performed.

The `PersistenceManager.newQuery()` method will instantiate a `javax.jdo.Query` object. The set of candidate objects to be returned by the query is then configured using the `setCandidates()` method. This is overridden to accept either a `java.util.Collection` or a `javax.jdo.Extent` (more on `Extent` shortly). A filter can also be defined using `setFilter()`, as can parameters to the filter. Variables are used to declare iterators over multi-valued collection fields.

FIGURE 8.14

Classes and interfaces that support JDO queries.



The `Extent` interface is principally used to identify all instances of some specified class in the persistent data store. Conceptually, this could be a very large set of objects, so commercial JDO implementations are not expected to fully materialize the set. Indeed, the JDO specification explicitly requires that JDO implementations must not cause an out-of-memory error when instantiating an `Extent`. The `java.util.Iterator` returned by the `iterator()` method must be able to iterate over all instances if needed, but notice that other methods of the `java.util.Collection` (such as `contains()` and `isEmpty()`) are not present in the `Extent` interface. In practice, the `Extent` and `Query` implementations will be closely coupled to provide efficient ways to identify the qualifying persistence-capable objects.

As an example, Listing 8.11 shows two queries. The first identifies all jobs that have a customer of `"winston"`; the second finds all jobs that require `"Cigar Trimmer"` as a skill.

LISTING 8.11 Using JDO to Search for Jobs that Meet Some Criteria

```
1: // import javax.jdo.*;
2: // import javax.naming.*;
3:
4: Context ctx = new InitialContext();
```

LISTING 8.11 Continued

```
 5: PersistenceManagerFactory pmf = (PersistenceManagerFactory)
 6:     ctx.lookup("java:comp/env/jdo/SomePersistenceManagerFactory";
 7: PersistenceManager pm = pmf.getPersistenceManager();
 8:
 9: Transaction txn = pm.currentTransaction();
10: txn.begin();
11:
12: // query #1: all jobs that have a customer of "winston".
13: Query query1 = pm.newQuery();
14: query1.setClass(Job.class);
15: Extent candidateJobs = pm.getExtent(Job.class, false);
16: query1.setCandidates(candidateJobs);
17: query1.declareParameters("String nameParam");
18: query1.setFilter( "customer == nameParam ");
19: Collection query1Res = (Collection) query.execute("Winston");
20: for(Iterator iter = query1Res.iterator(); iter.hasNext(); ) {
21:     Job job = (Job)iter.next();
22:     System.out.println(job.getRef());
23: }
24:
25: // query #2: all jobs that require "Cigar Trimmer" as a skill
26: Query query2 = pm.newQuery(Job.class, candidateJobs);
27: query2.declareVariables("Skill eachSkill");
28: query2.setFilter(
    ➥"skills.contains(eachSkill) &&
    ➥eachSkill.name == \"Cigar Trimmer\"");
29: Collection query2Res = (Collection) query.execute();
30: for(Iterator iter = query2Res.iterator(); iter.hasNext(); ) {
31:     Job job = (Job)iter.next();
32:     System.out.println(job.getRef());
33: }
34: txn.commit();
```

Other Features

There is much more in JDO than there is room to cover here. Some aspects worth a brief mention include the following:

- *Deployment descriptors*—A deployment descriptor is used for each persistence-capable application class. This identifies the fields that are persistent rather than transient, the JDO identity type, and other information required for the JDO enhancer.
- *Second Class Objects (SCOs)*—These are similar in concept to dependent value classes in EJBs, in that they are persistent only by virtue of being reachable from *First Class Object*s (all objects previous discussed have been First Class Objects). In particular, they do not have a JDO identity. If an SCO is modified, it must explicitly notify its containing First Class Object.

The JDO specification identifies various classes in the JDK library packages that are first- or second-class, or that are not persistable at all (for example, `java.net.Socket`).

- *Lifecycle*—Persistent-capable objects go through different stages of their lifecycle. For example, when an object is first instantiated, it is in the *transient* state. When the `PersistenceManager.makePersistent()` method is called, the object transitions to *persistent-new* state. One state, *hollow*, is very similar to the EJB notion of a passivated Entity bean.
- *Transient transactional objects*—These are persistent-capable objects acting in a manner akin to a `ShoppingCart` Session bean whose implementation supports automatic transaction recovery (implementing `javax.ejb.SessionSynchronization`).

Gotchas

The following are some "gotchas" to help you with your implementation:

- If writing CMTD Session beans, resource manager transaction methods (such as `connection.commit()`) must not be used.
- The `ejbCreate()` and `ejbRemove()` lifecycle methods for Session beans (and Message-driven beans) are performed with unspecified transaction context.
- The collection returned by the getter method for a cmr-field cannot be used outside of the transaction in which it was materialized (see EJB specification, section 10.3.8).
- If using a SQLj Part 0 compiler, make sure that the vendor offers adequate support for the generated code to run within the J2EE environment.
- If using SQLj Part 1 or Part 2, be sure to do some performance benchmarking first. This is a new technology and JVM-enabled features within the RDBMS are highly unlikely to perform as well as raw SQL.
- If the `javax.transaction.UserTransaction` interface is used for transaction demarcation within BMTD Session beans that are using JDO for persistence, the `javax.jdo.PersistenceManager` must be acquired after the transaction has been started.

  Moreover, the `PersistenceManager` cannot be used after the transaction has complete. It is better to use the `javax.jdo.Transaction` that does not require a new `PersistenceManager` for each new transaction.
- One for you to revisit after Day 9, "Java Messaging Service," and Day 10, "Message-Driven Beans"—the JMS request/reply paradigm cannot be used for transacted sessions (see EJB specification, section 17.3.5).

Summary

Curious day, today. If you've skipped over all the sections that discuss technologies that aren't appropriate to you, you might still be feeling pretty fresh. On the other hand, if you've been trying to wrap your head around all these new and peculiar concepts, you could be completely worn out.

Anyway, today you've learned that EJB containers support container-managed transaction demarcation, but that your Session beans can take control using bean-managed transaction demarcation if needed. Behind the scenes, there are some complex goings on with the XA interfaces to support distributed transactions, using 2PC and XA-aware data store connections, such as `javax.sql.XADataSource`.

JDBC is the de-facto way for implementing persistence of Java objects, and JDBC 3.0 unifies the various classes and interfaces in the `java.sql` and `javax.sql` packages. JDBC 2.1 and 3.0 have fleshed out support for the most relevant features of SQL1999, specifically advanced data types.

The SQLj initiative addresses both client-side integration of SQL and Java (part 0) and server-side (that is, RDBMS) integration (parts 1 and 2). SQLj part 0 is more succinct than JDBC, but is not particularly well tailored to the J2EE environment, because for the most part, it pre-dates it. SQLj parts 1 and 2 offer some opportunities to radically change the way that databases are used, with SQLj part 2 operating in much the same space as the SQL1999 advanced data types.

JDO is a new specification that aims to make Java persistence totally transparent. It is a natural fit for OODBMS and O/R mapping tool vendors and also works as a front-end to ERP systems. JDO primarily focuses on the API into the client-side cache of persistent objects, leaving much of the back-end configuration to the JDO implementation vendors.

Q&A

Q Why would it be pointless to deploy the getter of a multi-valued `cmr-field` using the `RequiresNew` transaction attribute?

A This would be pointless because the returned collection cannot be used outside of the transaction in which it was materialized.

Q What happens to the current transaction when an EJB throws an application exception?

A The bean may or may not mark the transaction for rollback; consult the bean's documentation.

Q When using SQLj Part 2, why should the Java class acting as an abstract data type define a static variable `serialVersionUID`?

A To provide forward compatibility with future versions of the class.

Exercises

A nice short exercise for you today.

The starting point for the exercise is a version of the today's case study that uses the BMTD for the `AdvertiseJob` bean. Convert the `Register` bean (that deals with applicants) to use BMTD rather than CMTD. The code to work on is under `day08\exercise`.

The steps you should perform are as follows:

1. Make sure that you are still using the CMP version of the `Agency` database. Although you won't be changing them, the Entity EJBs for the exercise are the CMP versions from yesterday.
2. Update the implementation of the various methods of `RegisterBean` class, basing it on `AdvertiseJobBean`. You will probably find it helpful to copy over the 3 helper methods (`beginTransactionIfRequired()`, `statusAsString()`, and `completeTransactionIfRequired()`).
3. Modify the `dd\agency_session_ejbs-ejb-jar.xml` deployment descriptor, changing `RegisterBean` to use `transaction-type` of `Bean`. You can also remove the `RegisterBean`'s `container-transaction` elements under the `assembly-descriptor` element because they will no longer be needed.
4. Build the enterprise application (`agency.ear` in the `jar` directory) and then load it into `deploytool`. Deploy and complete any information missing from the wizard.
5. Test your program using the `AllClients` client, run with `run\runAll`.

The solution is in `day08\agency`.

WEEK 2

DAY 9

Java Message Service

So far, you have learned about two types of EJB—Entity Beans and Session Beans. Before introducing you to the third type—Message Beans (to be covered on Day 10, "Message-Driven Beans"), we will take a look at the purpose and use of messaging within enterprise applications, the Java Message Service's API (JMS), and its place within J2EE.

Messaging

Many enterprise applications are built from separate software components. These components can reside on the same system or in a distributed or multi-tiered environment on several Java Virtual Machines.

Some method of communication is almost always required between the components in large systems. It is often a requirement that this communication should also be loosely coupled or asynchronous. In asynchronous communication, the sender sends the message and continues execution without waiting for a reply, and the receiver can retrieve the message at any time after it has been sent. In a loosely coupled system, the sender does not need to necessarily know who the recipient is; the communication itself may or may not be asynchronous. This

communication between software components is called *messaging*.

Without support for a messaging system, the programmer would typically use a sockets interface for inter-application communication. With sockets, both the sender and receiver need to agree on the socket address, and both applications need to be running at the same time. Likewise, if Remote Method Invocation (RMI) is used, the sender (or calling application) needs to know about the receiver's (or remote application's) methods. With messaging, the sender and receiver only have to agree on the message format and where to send it. The sender and receiver do not need to know anything about each other, nor do they both need to exist at the same time.

Messaging should be used in preference to a tightly coupled API such as RMI when some or all of the following conditions exist:

- The components interfaces are not known or not published.
- Not all of the components will be running at the same time.
- A sender needs to communicate with multiple receivers.
- No response is required.

Messaging should not be confused with electronic mail (e-mail). E-mail is used to communicate between people, whereas messaging is the mechanism used to communicate between applications.

Messaging is used in various environments and many applications; it is not unique to J2EE. A well-known example of a messaging service is IBM's MQ Series. While there are numerous messaging APIs, the only one supported by J2EE is JMS.

Message Passing

There are several models of message passing, but JMS only supports two—point-to-point and publish/subscribe. Both of these are known as a *push* models. The sender of the message is the active initiator, and the receiver is a passive consumer.

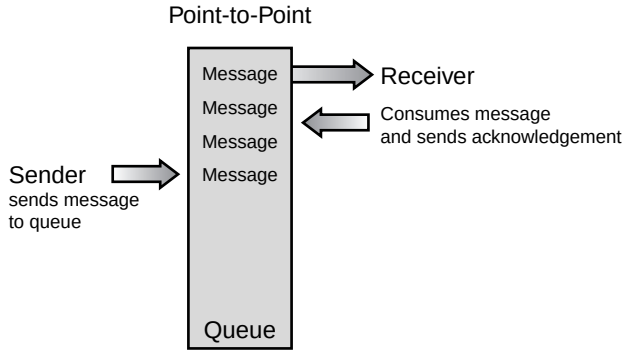
With point-to-point, the sender and receiver agree on a message destination, also known as a *queue*. The sender leaves the message and the receiver picks up the message at any time thereafter. The message remains in the queue until the receiver removes it. The following are some real-life examples of point-to-point communication:

- Sending a fax
- Dropping letters at a hotel desk to be picked up later by a hotel guest
- Leaving a voice mail message

Figure 9.1 illustrates point-to-point messaging.

FIGURE 9.1

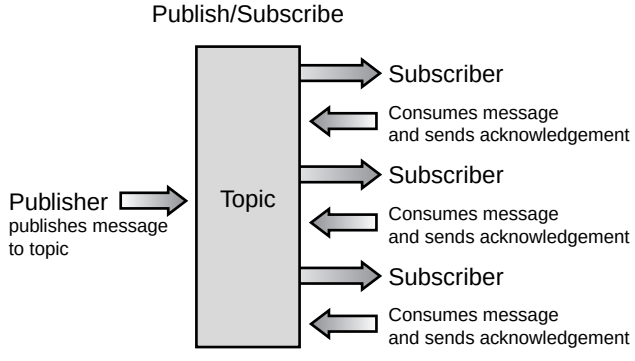
Point-to-point messaging model.



With the publish/subscribe model, the sender, now called a publisher, again sends messages to an agreed destination. The destination is, by convention, known as a topic and this time there may be many receivers and these are called *subscribers*. Messages are immediately delivered to all current subscribers and are deleted when all the subscribers have received the message. Figure 9.2 portrays the publish/subscribe messaging model.

FIGURE 9.2

Publish/Subscribe messaging model.



Within JMS, these models are called *message domains*. Code examples and further details on the JMS implementation of these message domains are presented in the next section.

Java Message Service API

The JMS API is a Message-Oriented Middleware (MOM) API. It was designed by a collaboration of several companies, including Sun Microsystems and IBM. Version 1 of the API was released in August, 1998. The purpose of JMS is to enable applications to transmit and receive messages in an asynchronous and reliable way. The JMS API also

defines interfaces to allow applications written using JMS to communicate with other messaging APIs.

The JMS designers' aims were as follows:

- To minimize the messaging concepts that a programmer needs to understand
- Provide communication that is loosely coupled, asynchronous, and reliable
- Have a consistent API that is independent of the JMS provider
- Maximize the portability of JMS applications

The JMS API provides messaging that is

- *Loosely coupled*—The sender and receiver may have no information about each other or the mechanisms used to process messages.
- *Asynchronous*—Receivers do not request messages. Messages can be delivered as they arrive. The sender does not wait for reply.
- *Reliable*—A message is sent and received once and only once.

Note the JMS API does not specify how to control the privacy and integrity of messages. The type and level of message security is left to the JMS providers and is configured by administers not by J2EE clients.

JMS and J2EE

Prior to J2EE 1.3, it was not necessary to implement the JMS API. From version 1.3, the JMS API became an integral part of the J2EE requirements.

Support for JMS within the J2EE platform provides the following functionality and features:

- Message beans for the sending and receiving of asynchronous messages (message beans are covered on Day 10)
- Support for distributed transactions
- Concurrent consumption of messages

An additional goal is to provide loosely coupled, reliable, asynchronous communication with legacy systems.

Programmatically, JMS can be considered to be a container-managed resource, similar to a JDBC connection. Application clients, EJBs, and Web components can all utilize the JMS API to send and receive messages. Note, however, that Applets are not required to support JMS.

The addition of the JMS API to J2EE simplifies enterprise software development and provides access to existing Enterprise Information Systems (EIS) and other legacy systems.

JMS API Architecture

Before launching into the JMS API programming model, there are some details of the JMS API architecture that need to be covered. Table 9.1 contains a list of the J2EE components that are found in a JMS application.

TABLE 9.1 JMS Components

Component	*Description*
JMS Provider	This is the messaging system that implements the JMS API interfaces and provides certain control features. All J2EE implementations from J2EE 1.3 and later must include a JMS provider.
JMS Clients	System components or programs that send or receive messages.
Messaging Domains	JMS supports both the point-to-point and publish/subscribe message domains. Both must be implemented in J2EE 1.3 and later.
Messages	These are the objects used to communicate between JMS clients.
Queues	Used to hold messages in the point-to-point domain.
Topics	Used to hold messages in the publish/subscribe domain.
Administered Objects	These are pre-configured JMS objects used to create connections with a JMS Provider (a connection factory) or to specify the target or source of messages (queues or topics).
Connection	A client's active connection to the JMS provider.
Session	One or more sessions can be created for each connection. Used to create senders and receivers and administer transactions.
Native Clients	Clients that use a non-JMS messaging API.

Figure 9.3 is a UML activity diagram of how the components fit together in a JMS application.

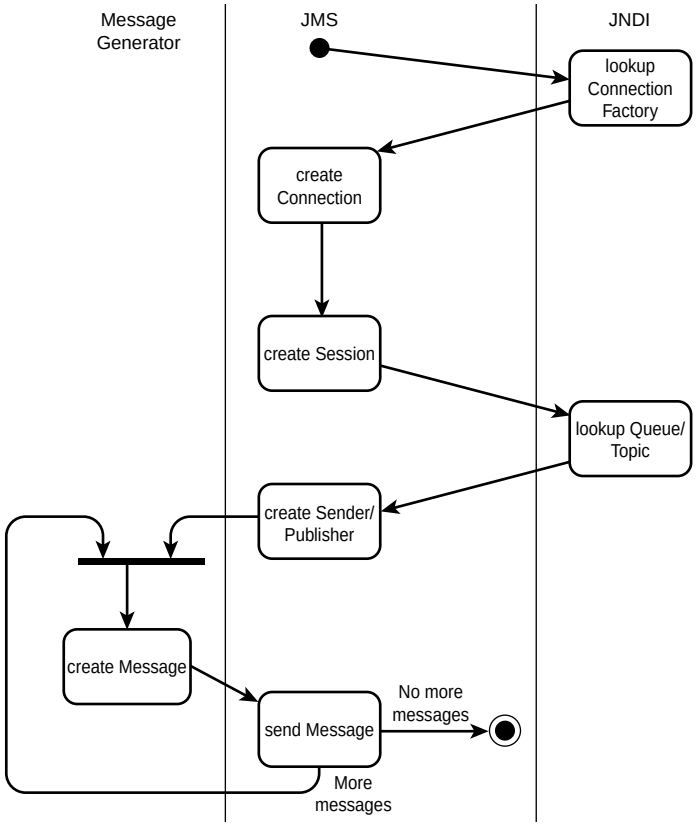


FIGURE 9.3
JMS application components.

Message Domains

JMS supports two types of message domains—point-to-point and publish/subscribe, giving application developers both choice and flexibility in message handling.

JMS Point-to-Point Messaging Domain

With point-to-point messaging the application is built around message queues, with a one-to-one relationship between a sender and a receiver. Each sender addresses the message to, and the receiver removes messages from, a queue. Many receivers can access the same queue, but only the first to pick up the message will receive it. The sender may set an expiration time on a message, after which it will be deleted from the queue.

There is no mechanism to send a message to a particular receiver. If this one-to-one relationship is required, the message must be sent to a queue that has only one receiver.

A JMS point-to-point message domain has the following characteristics:

- Each message is produced by the sender and consumed by one and only one receiver.
- Messages are either consumed by the receiver or they time out and are deleted by the JMS provider.
- Receivers can consume the message any time after it has been sent.
- The receiver does not need to exist when the message is produced.
- The receiver cannot request a message.
- The receiver must acknowledge receipt of the message.

The JMS API does not allow a single message to be sent to multiple queues. If you need to send a message to multiple receivers, you should consider using the publish/subscribe message domain.

JMS Publish/Subscribe Messaging Domain

The JMS publish/subscribe messaging domain has a completely different delivery model from point-to-point. With the publish/subscribe model, senders post messages to a topic; many receivers can register interest in a topic by subscribing to it.

Messages in a topic are distributed to all subscribers and only exist in the topic for as long as this process takes. A receiver can only receive messages posted after it has subscribed to a topic, and the receiver must be active at the time the message is posted to receive it. This introduces a timing dependency between the sender and the receivers that is not present in point-to-point messaging.

To circumvent this timing dependency, the JMS API allows receivers to create durable subscriptions. Durable subscriptions will be covered in more detail in the section titled "Creating Durable Subscriptions," later on in this chapter.

A JMS publish/subscribe message domain has the following characteristics:

- Each message is produced by a publisher and consumed by zero or more subscribers.
- Messages are immediately distributed to the existing subscribers.
- Subscribers must exist at the time the message is published to receive the message.
- Durable subscriptions can be used to allow subscribers to receive messages published to a topic when the subscriber was inactive.
- The subscriber must acknowledge receipt of the message.

Table 9.2 lists the JMS objects in the point-to-point domain.

TABLE 9.2 Components in the JMS Point-to-Point Message Domain

Component	*Description*
`QueueConnectionFactory`	Administered object used to create an object that implements the `QueueConnection` interface
`QueueConnection`	Active connection to a JMS provider
`QueueSession`	Provides methods for creating objects that implement the `QueueSender` `QueueReceiver` or `QueueBrowser` interfaces
`QueueSender`	Used by a client to send a message to a queue
`QueueReceiver`	Used to receive messages sent to a queue
`Queue`	Encapsulates the JMS provider queue name
`QueueBrowser`	Used to look at messages on a queue without removing them

Developing JMS Applications Using JBoss[1]

Shortly, you will be developing a simple JMS application. As with all the code examples in this book, the application will be developed using J2EE RI. Although J2EE RI provides a good environment for learning about JMS, it is not intended to support commercial applications. Before developing this JMS application, there will be a short digression to show the JMS features available with JBoss.

JBoss is in an open-source implementation of J2EE developed by the JBoss Group. JBoss is available to download for free from `www.jboss.org`.

JMS Implementation in JBoss

JBoss contains a JMS provider called JBossMQ. JBossMQ is fully JMS API-compliant and can be used for standalone JMS clients. JBossMQ has a number of configuration files, the most important of which is called `jboss.jcml`. This file resides under the JBoss installation in the `conf/default` directory. To create a message application in Jboss, you first have to read or edit this file to

1. Look up the names of the available connection factories
2. Add destinations (queues and topics)

Additionally, you can optionally add client identifiers (JBossMQ users) to the `jbossmq-state.xml` file (also in `conf/default`) for authentication. The purpose and use of authentication is to associate a client connection and its objects with a state maintained by the JMS provider. At the current time, the only state defined is that required to support durable subscriptions.

Administered Objects

Connection factories and destinations are the administered objects that are configured by the JMS administrator.

Connection Factory

The purpose of a connection factory is to encapsulate connection configuration. Clients use a pre-configured connection factory to create connections to the JMS provider.

JBoss includes several different connection factories with varying characteristics, as shown in Table 9.3.

TABLE 9.3 Connection Factories Provided in JBoss

JNDI Name	*Description*
`ConnectionFactory`	The default factory. A fast two-way socket-based communication protocol.
`XAConnectionFactory`	Also a fast two-way socket based protocol that also has support for XA transactions.
`RMIConnectionFactory`	Uses RMI to implement communication mechanism.
`RMIXAConnectionFactory`	Uses RMI and also has support for XA transactions.
`java:/ConnectionFactory`	Very fast in-VM protocol that does not use sockets. Available when the client is in the same virtual machine as JBossMQ.
`java:/XAConnectionFactory`	Fast in-VM protocol that also. supports XA transactions.
`UILConnectionFactory`	Protocol multiplexed over one socket. Used when going through firewalls or when the client is not able to look up the server IP address correctly.
`UILXAConnectionFactory`	Same as `UILConnectionFactory` but also has support for XA transactions.

You should choose the connection factory appropriate for your application. The default connection factory in JBossMQ is `ConnectionFactory`, which is implemented over a two-way, socket-based communication protocol.

Note that other JMS providers may have different connection factories with different characteristics and different names.

Destinations

Destinations define the address parameters for queues and topics. New queue and topic destinations are added to the `jboss.jcml` file with the following:

```
<mbean code="org.jboss.mq.server.TopicManager"
➥ name="JBossMQ:service=Topic,name=YourTopicName"/>
<mbean code="org.jboss.mq.server.QueueManager"
➥ name="JBossMQ:service=Queue,name=YourQueueName"/>
```

You have now finished looking at JBoss. The following examples illustrate how to connect to queues and topics using J2EE RI.

Programming a JMS Application Using J2EE RI

In this section, you will develop a simple point-to-point application that sends and receives a text message. Although this code has not been written for any one particular JMS implementation, you will need to be aware that the mechanism for setting up the administered objects is implementation specific. Also, the JNDI names may be different for different J2EE environments.

J2EE RI Connection Factories

With the J2EE RI, you will use the default connection factory objects, named `jms/QueueConnectionFactory` and `jms/TopicConnectionFactory`.

Adding Destinations in J2EE RI

J2EE RI provides a command-line program called `j2eeadmin` to list, create, modify, and delete queues and topics. There is also a screen in the J2EE RI Application Deploy Tool that provides an interface to `j2eeadmin` for the same purpose.

Creating a Queue in J2EE RI

The point-to-point application uses a queue for communication. To create a queue do the following:

1. Ensure that J2EE server is running.
2. Use `j2eeadmin` or `deploytool` to create the queue.

Create a Queue Using `j2eeadmin`

To see the existing queues and topics, use the following:

```
j2eeadmin -listJMSDestination
```

The J2EE RI has a default queue predefined—`jms/Queue`.

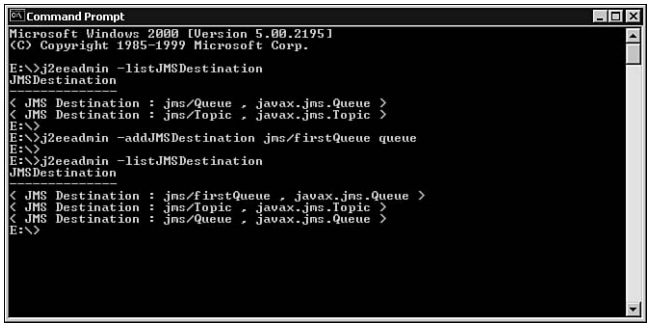
To add a queue called `jms/firstQueue`, use the following:

```
j2eeadmin -addJMSDestination jms/firstQueue queue
```

The `j2eeadmin` command works silently, so to check that your queue has been created, run `J2eeadmin -listJMSDestination` once more. Figure 9.4 demonstrates the use of these two commands to create a queue called `jms/firstQueue` (this is the queue you will use in your first example).

FIGURE 9.4

Using `j2eeadmin` to add a JMS queue to J2EE RI.

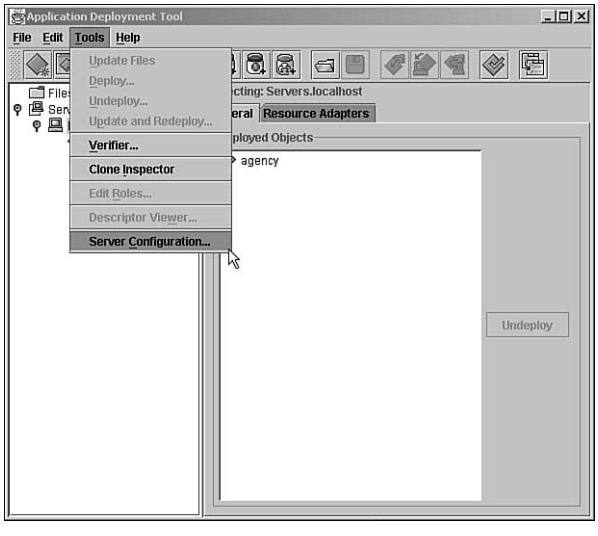


Create a Queue Using `deploytool`

You can also use `deploytool` to list and add queues. Start up `deploytool` and select Server Configuration from the Tools menu, as shown in Figure 9.5.

FIGURE 9.5

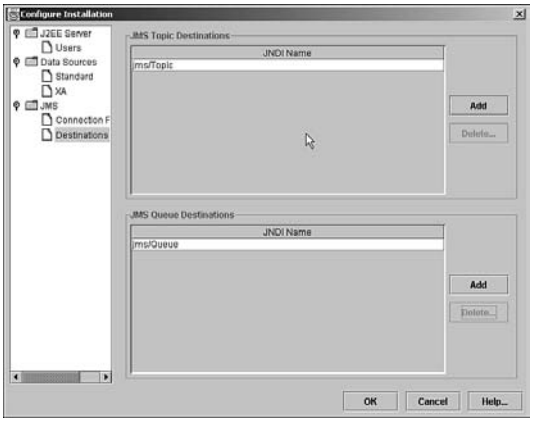
Using `deploytool` to configure installation.



This will bring up the Configure Installation screen. Select JMS Destinations in the panel on left to display the screen in Figure 9.6.

FIGURE 9.6

Adding JMS destination in `deploytool`.



Click the Add button in the panel at bottom right titled JMS Queue Destinations and add a new queue called **`jms/firstQueue`**.

Point-to-Point Messaging Example

You are now in a position to start coding. For all the following code examples, you will need to import the `javax.jms` package. This contains all the classes for creating connections, sessions, queues, and topics.

To send a JMS message, a number of steps must first be performed to obtain a connection factory, establish a session, and create a `QueueSender` object. Although the example is in the point-to-point message domain, the same steps are required for publish/subscribe topics.

The following steps show how to create a queue used later to send a simple text message.

1. Obtain the JNDI initial context.

   ```
   Context context = new InitialContext();
   ```

2. Contact the JMS provider, obtain a JMS connection from the appropriate `ConnectionFactory`, and create a connection for a queue. The following code uses a connection factory registered against the default JNDI name `jms/QueueConnectionFactory`.

   ```
   QueueConnectionFactory queueFactory =
   ➥ (QueueConnectionFactory)context.lookup("jms/QueueConnectionFactory");
   QueueConnection queueConnection = queueFactory.createQueueConnection();
   ```

The `createQueueConnection()` method throws a `JMSException` if the JMS provider fails to create the queue connection due to some internal error.

3. Establish a `QueueSession` for this connection. In this case, the `QueueSession` has transactions set to `false` (transactions are covered later) and `AUTO ACKNOWLEDGE` of receipt of messages. This will throw a `JMSException` if the `QueueConnection` object fails to create a session.

```
QueueSession queueSession = queueConnection.createQueueSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
```

> Although sent messages are not acknowledged, a session can be used to receive messages created by its own connection. This is why an Acknowledge mode must be specified on a session, even if the queue is only used to send messages.

4. Obtain the queue destination using its JNDI name as defined using `j2eeadmin` or `deploytool`. The `lookup()` method can throw a `NamingException` if the name is not found.

```
queue = (Queue)context.lookup("jms/firstQueue");
```

5. Finally, create a queue sender that will be used to send messages. This will throw a `JMSException` if the `session` fails to create a sender, and an `InvalidDestinationException` if an invalid queue is specified.

```
QueueSender queueSender = queueSession.createSender(queue);
```

Note how the connection factory hides all the implementation details of the connection from the client. It does the hard work of creating resources, handling authentication, and supporting concurrent use.

Now you have a queue that is ready to send messages. But before that, you need to know a little more about JMS messages.

JMS Messages

JMS messages consist of three parts:

- *A header*—Used to identify messages, set priority and expiration, and so on and to route messages.
- *Properties*—Used to add additional information in addition to the message header.
- *Message body*—There are five message body forms defined in JMS—`BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage`, and `TextMessage`.

Note that only the header is a required component of a message; the other two parts, including the body, are optional.

Message Header Fields

The JMS message header contains a number of fields that are generated by the JMS provider when the message is sent. These include the following:

- `JMSMessageID` A unique identifier
- `JMSDestination` Represents the queue or topic to which the message is sent
- `JMSRedelivered` Set when the message has been resent for some reason

The following three header fields are available for the client to set:

- `JMSType` A string that can be used to identify the contents of a message
- `JMSCorrelationID` Used to link one message with another, typically used to link responses to requests
- `JMSReplyTo` Used to define where responses should be sent

Other header fields may be set by the client but can be overridden by the JMS provider with figures set by an administrator:

- `JMSDeliveryMode` This can be either `PERSISTENT` or `NON_PERSISTENT` (the default is `PERSISTENT`).
- `JMSPriority` Providers recognize priorities between 0 and 9, with 9 being the highest (default is 4). Note that there is no guarantee that higher priority messages will be delivered before lower priority ones.
- `JMSTimestamp` This contains the time the message was sent to the JMS provider by the application. Note that this is not the time the message is actually transmitted by the JMS provider to the receivers.
- `JMSExpiration` An expiration time

Each header field has associated setter and getter methods that are fully described on the JMS API documentation.

Message Properties

As you have seen, there is not a great deal of scope for clients to add information to JMS header fields, but additionally JMS messages can incorporate properties. These are name/value pairs defined by the client.

Property values can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `String` and are defined using the appropriate `Message.setProperty` method. For example:

```
message.setStringProperty ("Type", "Java");
```

sets the message object's property called `"Type"` to the string `"Java"`. A corresponding `getProperty` method is used to retrieve a message's property by name. The `getPropertyNames()` can be used if the names are not known.

The prefix JMSX is used for JMS-defined properties. Inclusion of these JMSX properties are optional; refer to your JMS provider documentation to determine what properties are supported in your JMS implementation.

JMS Body Types

JMS supports five types of message body. Each is defined by its own message interface. Each type is only briefly described in Table 9.4. Refer to the JMS API for more information on the message body types.

TABLE 9.4 JMS Message Body Types

Message Body Type	*Message Contents*
`BytesMessage`	Un-interpreted byte stream.
`MapMessage`	Name/value pairs
`ObjectMessage`	Serializable Java object
`StreamMessage`	Stream of Java primitives
`TextMessage`	Java String

Creating a Message

For the example given here, the default header properties will be used. Also, to keep this example straightforward, a `TextMessage` body will be used (this is also the most common message body type). A `TextMessage` object extends the `Message` interface and is used to send a message containing a `String` object.

A text message body is created using the `createTextMessage()` method in the `QueueSession` object.

```
TextMessage message = queueSession.createTextMessage(String);
```

The content of the a text message is added with `message.setText()`:

```
String msg = "some text";
message.setText(msg);
```

Having created a message, you are ready to send it.

Sending a Message

A message must be sent to a queue `QueueSender` object as follows:

```
queueSender.send(queue, message);
```

It's as simple as that.

Closing the Connection

Connections are relatively heavyweight JMS objects, and you should always release the resources explicitly rather than depending on the garbage collector. The sender, the session, and the connection should all be closed when no more messages need to be sent.

```
queueSender.close();
queueSession.close();
queueConnection.close();
```

With the J2EE RI implementation, prior to releasing the `QueueSender` object, you must send a empty message to indicate no more messages will be sent. To do this, add the following line before the `QueueSender` object is closed.

```
queueSender.send(queueSession.createMessage());
```

This code may not be required in other JMS implementations.

Send JMS Text Message Example

The code for the entire point-to-point sender example is shown in Listing 9.1.

LISTING 9.1 Complete Listing for Point-to-Point Queue Sender

```
 1: import javax.naming.*;
 2: import javax.jms.*;
 3:
 4: public class PTPSender {
 5:     private QueueConnection queueConnection;
 6:     private QueueSession queueSession;
 7:     private QueueSender queueSender;
 8:     private Queue queue;
 9:
10:     public static void main(String[] args) {
11:     try {
12:          PTPSender sender = new PTPSender();
13:
14:         System.out.println ("Sending message Hello World");
15:         sender.sendMessage("Hello World");
16:         sender.close();
17:      } catch(Exception ex) {
18:         System.err.println("Exception in PTPSender: " + ex);
19:     }
```

LISTING 9.1 Continued

```
 20: }
 21:
 22:     public PTPSender() throws JMSException, NamingException {
 23:         Context context = new InitialContext();
 24:         QueueConnectionFactory queueFactory =
➥ (QueueConnectionFactory)context.lookup("jms/QueueConnectionFactory");
 25:         queueConnection = queueFactory.createQueueConnection();
 26:         queueSession = queueConnection.createQueueSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
 27:         queue = (Queue)context.lookup("jms/firstQueue");
 28:         queueSender = queueSession.createSender(queue);
 29:     }
 30:
 31:     public void sendMessage(String msg) throws JMSException {
 32:         TextMessage message = queueSession.createTextMessage();
 33:         message.setText(msg);
 34:         queueSender.send(message);
 35:     }
 36:
 37:     public void close() throws JMSException {
 38:          //Send a non-text control message indicating end of messages
 39:         queueSender.send(queueSession.createMessage());
 40:         queueSender.close();
 41:         queueSession.close();
 42:         queueConnection.close();
 43:     }
 44:}
```

To send messages to the queue, run this program from the command line. The next example program will retrieve messages from that same queue for display onscreen.

Consuming Messages

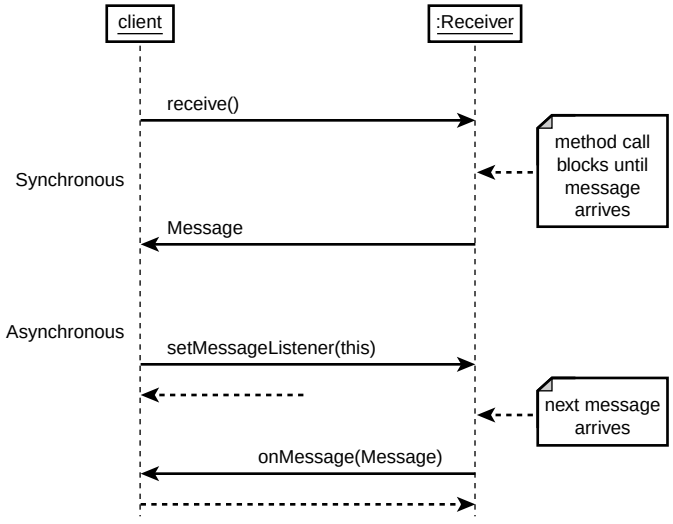
There are two ways of consuming queued messages with JMS.

- *Synchronously*—An explicit call to the `receive()` method.
- *Asynchronously*—By registering an object that implements the `MessageListener` interface. The JMS provider invokes the `onMessage()` method in this object each time there is a message queued at the destination.

Figure 9.7 is a UML sequence diagram showing the difference between synchronous and asynchronous message consumption.

FIGURE 9.7

Synchronous and asynchronous message consumption.



The following example will demonstrate how to receive a message from a queue using a synchronous call.

Simple Synchronous Receiver Example

The code for a simple synchronous receiver is very similar to that of the sender method already presented. There are two differences in constructing the `QueueReceiver` object.

The first one is obvious; a `QueueReceiver` is created instead of a `QueueSender`. Like `createSender()`, the `createReceiver()` method throws a `JMSException` if the session fails to create a receiver, and an `InvalidDestinationException` if an invalid queue is specified.

The second difference is that this time there is a call to the connection's `start()` method, which starts (or restarts) delivery of incoming messages for this receiver. Calling `start()` twice has no detrimental effect. It also has no effect on the connection's ability to send messages. The `start()` method may throw a `JMSException` if an internal error occurs. A receiver can use `Connection.stop()` to temporarily suspend delivery of messages.

The message is received using the synchronous `receive()` method, as shown next. This may throw a `JMSException`.

```
Message msgBody =queueReceiver.receive();
if (msgBody instanceof TextMessage) {
    String text = ((TextMessage) msgBody).getText();
}
```

If there is no message in the queue, the `receive()` method blocks until a message is available. There are two alternative versions of the `receive()` method:

- `receiveNoWait()` Retrieves the next message available, or returns `null` if one is not immediately available
- `receive(long timeout)` Retrieves the next message produced within the period of the timeout

Receive JMS Text Message Example

The code for the entire point-to-point receiver example is shown in Listing 9.2.

LISTING 9.2 Complete code for Point-to-Point Queue Receiver

```
 1: import javax.naming.*;
 2: import javax.jms.*;
 3:
 4: public class PTPReceiver {
 5:
 6:     private QueueConnection queueConnection;
 7:     private QueueSession queueSession;
 8:     private QueueReceiver queueReceiver;
 9:     private Queue queue;
10:
11:     public static void main(String[] args) {
12:         try {
13:             PTPReceiver receiver = new PTPReceiver();
14:             System.out.println ("Receiver running");
15:             String textMsg = receiver.consumeMessage();
16:             if (textMsg != null)
17:                 System.out.println ("Received: " + textMsg);
18:             receiver.close();
19:         }
20:         catch(Exception ex) {
21:              System.err.println("Exception in PTPReceiver: " + ex);
22:         }
23:     }
24:
25:     public PTPReceiver() throws JMSException, NamingException {
26:         Context context = new InitialContext();
27:         QueueConnectionFactory queueFactory =
➥ (QueueConnectionFactory)context.lookup("jms/QueueConnectionFactory");
28:         queueConnection = queueFactory.createQueueConnection();
29:         queueSession = queueConnection.createQueueSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
30:         queue = (Queue)context.lookup("jms/firstQueue");
31:         queueReceiver = queueSession.createReceiver(queue);
32:         queueConnection.start();
33:     }
```

LISTING 9.2 Continued

```
34:
35:     public String consumeMessage () throws JMSException {
36:         String text = null;
37:         Message msgBody = queueReceiver.receive();
38:         if (msgBody instanceof TextMessage) {
39:             text = ((TextMessage) msgBody).getText();
40:         }
41:         return text;
42:     }
43:
44:     public void close() throws JMSException {
45:         queueReceiver.close();
46:         queueSession.close();
47:         queueConnection.close();
48:     }
49: }
```

Run this program from the command line. If you have previously used `PTPSender` to put messages in the queue, these will now be displayed.

Asynchronous Messaging

For many applications, the synchronous mechanism is not suitable and an asynchronous technique is required. To implement this in JMS, you need to register an object that implements the `MessageListener` interface. The JMS provider invokes this object's `onMessage()` each time a message is available at the destination.

The receiver example will now be extended to support asynchronous messaging. Because this is a simple example, the listener is implemented in the same class.

```
public class PTPListener implements MessageListener {
```

The message listener is registered with a specific `QueueReceiver` by using the `setMessageListener()` method before calling the connection's `start()` method. The following extra line is required in the constructor:

```
queueReceiver.setMessageListener(this);
```

Messages might be missed if you call `start()` before you register the message listener.

To actually receive the messages, the `MessageListener` interface provides a single `onMessage()` method. The JMS provider calls your implementation of this method when it has a message to deliver. The following is an example `onMessage()` method:

```
public void onMessage(Message message) {
    try {
        if (message instanceof TextMessage) {
```

```
            String text = ((TextMessage) message).getText();
            System.out.println("Received: " + text);
        }
    }
    catch(JMSException ex) {
        System.err.println("Exception in OnMessage: " + ex);
    }
}
```


The `onMessage()` method should handle all exceptions. If `onMessage()` throws an exception, the signature will be altered and, therefore, not recognized.

In the `main()` method, the `QueueReceiver` object is initialized as before, but this time there is no call to a synchronous `receive()` method.

```
public static void main(String[] args) {
    System.out.println ("Listener running");
    try {
        PTPListener receiver = new PTPListener();
        Thread.currentThread().sleep(2000);
        receiver.close();
    }
    catch(Exception ex) {
        System.err.println("Exception in PTPListener: " + ex);
    }
}
```

A sleep has been placed in the main body to allow time for messages to be handled.

Normally, the `main()` method would sleep for a long period or do other processing and therefore requires some way to determine when message processing is finished. Typically, a shutdown message is sent or the user interface is used to close the application. For brevity, this code is not included here.

Asynchronous Exception Handling

If you register an `ExceptionListener` with a connection, your application will be asynchronously informed of problems. If a client only consumes a message, this may be the only way it can be informed that the connection has failed.

The JMS provider will call the listener's `onException()` method, passing it a `JMSException` describing the problem.

The Publish/Subscribe Message Domain

So far, you have seen how to create a simple point-to-point application with a single sender and receiver. Now you will build a simple bulletin board application to demonstrate the features of the publish/subscribe model.

For this example, a `TopicPublisher` will be produced that creates messages and publishes them to a bulletin board. A single `TopicSubscriber` will asynchronously listen for messages. The messages will be printed to the screen—one at time—in the order they were received. The program then exits.

Remember that, unlike messages in a queue, messages in a topic are immediately distributed to all subscribers. As a result, the timing of the publisher and the subscriber becomes important. Apart from this, the publisher/subscriber code is very similar to the sender/receiver code. In fact, the publisher code is a copy of the previous sender code with all references to `Queue<object>` changed to `Topic<object>`.

Table 9.5 lists the JMS objects in the publish/subscribe domain.

TABLE 9.5 Components in the JMS Publish/Subscribe Message Domain

Component	*Description*
`TopicConnectionFactory`	Administered object used to create an object that implements the `TopicConnection` interface
`TopicConnection`	Active connection to a JMS provider
`TopicSession`	Provides methods for creating objects that implement the `TopicPublisher` and `TopicSubscriber` interfaces
`TopicPublisher`	Used by a client to publish a message to a topic
`TopicSubscriber`	Used to receive messages sent to a topic
Durable `TopicSubscriber`	Used to receive messages published when the subscriber is inactive
`Topic`	Encapsulates the JMS provider topic name

Because subscribers only receive messages when they are active, it would be nice to be able to test for active subscribers before publishing to a topic. Unfortunately, this is not possible.

Publish/Subscribe Messaging Example

Now, you will build a simple bulletin board application. For this example, the bulletin board publisher program will generate 10 simple messages. The subscriber will be a Swing application that will display the messages as they arrive.

The bulletin board used a topic called `jms/bulletinBoard`. This must be created using the Configure Installation screen in `deploytool` or using `j2eeadmin` as follows:

```
j2eeadmin –addJMSDestination jms/bulletinBoard topic
```

Bulletin Board Publisher

The same mechanism is used to create a topic as a queue, so Listing 9.3 should appear very similar to that in the point-to-point receiver example, except that all references to a queue are replaced with topic.

LISTING 9.3 Bulletin Board Publisher Program


```
 1: import javax.naming.*;
 2: import javax.jms.*;
 3:
 4: public class BulletinBoardPublisher {
 5:     private TopicConnection topicConnection;
 6:     private TopicSession topicSession;
 7:     private TopicPublisher bulletinBoardPublisher;
 8:     private Topic bulletinBoard;
 9:
10:     public static void main(String[] args) {
11:         try {
12:             BulletinBoardPublisher publisher = new
BulletinBoardPublisher("TopicConnectionFactory","jms/bulletinBoard");
13:
14:             System.out.println ("Publisher is up and running");
15:
16:             for (int i = 0; i < 10; i++) {
17:                 String bulletin = "Bulletin Board Message number: " + i;
18:                 System.out.println (bulletin);
19:                 publisher.publishMessage(bulletin);
20:             }
21:             publisher.close();
22:         } catch(Exception ex) {
23:             System.err.println("Exception in BulletinBoardPublisher: " +
ex);
24:         }
25:     }
26:
27:     public BulletinBoardPublisher(String JNDIconnectionFactory,
➥ String JNDItopic) throws JMSException, NamingException {
28:         Context context = new InitialContext();
29:         TopicConnectionFactory topicFactory = (TopicConnectionFactory)con-
text.lookup(JNDIconnectionFactory);
30:         topicConnection = topicFactory.createTopicConnection();
31:         topicSession = topicConnection.createTopicSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
32:         bulletinBoard = (Topic)context.lookup(JNDItopic);
33:         bulletinBoardPublisher =
➥ topicSession.createPublisher(bulletinBoard);
34:     }
35:
```

LISTING 9.3 Continued

```
36:     public void publishMessage(String msg) throws JMSException {
37:         TextMessage message = topicSession.createTextMessage();
38:         message.setText(msg);
39:         bulletinBoardPublisher.publish(message);
40:     }
41:
42:     public void close() throws JMSException {
43:         bulletinBoardPublisher.close();
44:         topicSession.close();
45:         topicConnection.close();
46:     }
47: }
```

Run this program from the command line. This will check that this program runs okay, but remember that messages published to topics are not persistent. For the subscriber program to pick up the messages, you will need to run this program again while the subscriber is running.

Bulletin Board Subscriber

The subscriber is a Swing application that outputs the bulletins as they arrive (see Listing 9.4).

Remember that for this program to receive the bulletins, it must be running when they are published.

LISTING 9.4 Bulletin Board Subscriber Program

```
 1: import javax.naming.*;
 2: import javax.jms.*;
 3: import java.io.*;
 4: import javax.swing.*;
 5: import java.awt.*;
 6: import java.awt.event.*;
 7: public class BulletinBoardSubscriber extends JFrame
➥ implements MessageListener {
 8:     private TopicConnection topicConnection;
 9:     private TopicSession topicSession;
10:     private TopicSubscriber bulletinBoardSubscriber;
11:     private Topic bulletinBoard;
12:     private JTextArea textArea = new JTextArea(4,32);
13:
14:     public static void main(String[] args) {
15:         try {
16:             final BulletinBoardSubscriber subscriber = new
➥ BulletinBoardSubscriber("jms/TopicConnectionFactory","jms/bulletinBoard");
```

LISTING 9.4 Continued

```
  17:             subscriber.addWindowListener(new WindowAdapter() {
  18:                 public void windowClosing(WindowEvent ev) {
  19:                     try {
  20:                         subscriber.close();
  21:                     } catch(Exception ex) {
  22:                         System.err.println("Exception in
➥ BulletinBoardSubscriber: " + ex);
  23:                     }
  24:                     subscriber.dispose();
  25:                     System.exit(0);
  26:                 }
  27:             });
  28:             subscriber.setSize(500,400);
  29:             subscriber.setVisible(true);
  30:         } catch(Exception ex) {
  31:             System.err.println("Exception in BulletinBoardSubscriber: "
➥ + ex);
  32:         }
  33:     }
  34:
  35:     public BulletinBoardSubscriber(String JNDIconnectionFactory,
➥ String JNDItopic) throws JMSException, NamingException {
  36:         super (JNDIconnectionFactory+":"+JNDItopic);
  37:         getContentPane().add(new JScrollPane(textArea));
  38:         Context context = new InitialContext();
  39:         TopicConnectionFactory topicFactory = (TopicConnectionFactory)con-
text.lookup(JNDIconnectionFactory);
  40:         topicConnection = topicFactory.createTopicConnection();
  41:         topicSession = topicConnection.createTopicSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
  42:         bulletinBoard = (Topic)context.lookup(JNDItopic);
  43:         bulletinBoardSubscriber =
➥ topicSession.createSubscriber(bulletinBoard);
  44:         bulletinBoardSubscriber.setMessageListener(this);
  45:         topicConnection.start();
  46:     }
  47:
  48:     public void onMessage(Message message) {
  49:         try {
  50:             if (message instanceof TextMessage) {
  51:                 String bulletin = ((TextMessage) message).getText();
  52:                 String text = textArea.getText();
  53:                 textArea.setText(text+"\n"+bulletin);
  54:             }
  55:         } catch(JMSException ex) {
  56:              System.err.println("Exception in
➥ BulletinBoardSubscriber:OnMessage: " + ex);
  57:         }
```

LISTING 9.4 Continued

```
58:     }
59:
60:
61:     public void close() throws JMSException {
62:         bulletinBoardSubscriber.close();
63:         topicSession.close();
64:         topicConnection.close();
65:     }
66: }
```

When you run this program from the command line, a small window will appear. Any messages published to the bulletin board topic while the program is running will appear in this window.

Creating Durable Subscriptions

When you run the bulletin board example, you will have seen that you need to get the timing right and that the subscriber can miss bulletins if it is not running when they are sent. This is because the `TopicSession.createSubscriber()` method creates a non-durable subscriber. A non-durable subscriber can only receive messages that are published while it is active.

To get around this restriction, the JMS API provides a `TopicSession.createDurableSubscriber()` method. With a durable subscription, the JMS provider stores the messages published to the topic, just as it would store messages sent to a queue.

Figure 9.8 shows diagrammatically how messages are consumed with non-durable and durable subscriptions when the subscriber is inactive during the period when messages are published.

To create a durable subscription, you must associate a connection factory with a defined user and use this factory to create the connection. You will be shown how to create users on day 15, “Security,” but for now, you can use the user guest that has been set up for you.

Use the following command to associate a connection factory with the user `guest`:

```
j2eeadmin -addJmsFactory jms/DurableTopic topic -props clientID=guest
```

or you can add the factory using the Configure Installation screen in install tool. Select Connection Factories in the panel on the left. Add the new factory, `jms/DurableTopic`, to the panel on the top right and add `ClientID` as the Property Name with Value of `guest` in the panel at bottom right.

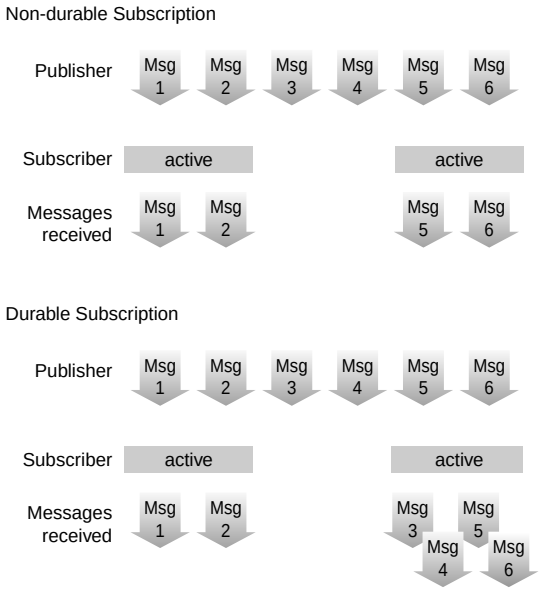


FIGURE 9.8
Non-durable and durable subscriptions.

After using this connection factory to create the connection and session, you call the `createDurableSubscriber` method with two arguments, the topic and the subscription ID string that specifies the name of the subscription:

```
String subID = "DurableBulletins";
TopicSubscriber topicSubscriber =
➥ topicSession.createDurableSubscriber(bulletinBoard, subID);
```

Messages are then read from the topic as normal. To temporarily stop receiving messages, you simply close the subscriber.

```
topicSubscriber.close();
```

Messages are now stored by the JMS provider until the subscription is reactivated with another call to `createDurableSubscriber()` with the same subscription ID.

A subscriber can permanently stop receiving messages by unsubscribing a durable subscription with the `unsubscribe()` method. You first need to close the subscriber.

```
topicSubscriber.close();
topicSession.unsubscribe(subID);
```

If you make these changes to the bulletin board subscriber program, you will not initially notice any difference in operation. The distinction becomes apparent if you close the subscriber program and run the publisher. Now when you start up the durable subscriber once more, you will receive messages sent to the bulletin board while the program was not running.

Additional JMS Features

The following sections cover some additional features available in JMS. Not all the features of JMS are covered, and you should refer to the JMS API specification for more information.

Message Selectors

So far, you have received all the messages sent. The JMS API provides support for filtering received messages. This is accomplished by using a message selector. The `createReceiver` and the two forms of `createSubscriber` (durable and nondurable) methods all have a signature that allows a message selector to be specified.

The message selector is a string containing an SQL conditional expression. Only message header values and properties can be specified in the message selector. Sadly, it is not possible to filter messages on the basis of the contents of the message body.

```
String highPriority = "JMSPriority = '9' AND topic = 'Java'";
bulletinBoardSubscriber = topicSession.createSubscriber(bulletinBoard,
➥ highPriority, false);
```

This selector will ensure that only priority nine messages are received. Note here that `topic` is a property of the message that has been created and set by the sender.

Notice that for this form of the `createSubscriber` the parameters are as follows:

```
TopicSession.createSubscriber(topic, messageSelector, noLocal);
```

You need to set the `noLocal` parameter to specify whether you want to receive messages created by your own connection. Set `noLocal` to `false` to prevent the delivery of messages created by the subscriber's own connection.

Session Acknowledgement Modes

In the examples given so far, auto acknowledgement has been used to send the acknowledgement automatically as soon as the message is received. This has the advantage of removing the burden of acknowledging messages from you, but it has the disadvantage that if your application fails before the message is processed, the message may be lost. After a message is acknowledged, the JMS provider will never redeliver it.

Deferring acknowledgement until after you have processed the message will protect against loss of data. To do this, the `session` must be created with client acknowledgement.

```
queueConnection.createQueueSession(false, Session.CLIENT_ACKNOWLEDGE);
```

Now when the message is received, no acknowledgement will be sent automatically. It is up to you to ensure that the message is acknowledged at some later point.

```
message = (TextMessage) queueReceiver.receive();
// process the message
message.acknowledge();
```

If you do not acknowledge the message, it may be resent.

A third acknowledgement mode, `DUPS_OK_ACKNOWLEDGE`, can be used when the delivery of duplicates can be tolerated. This is a form of `AUTO_ACKNOWLEDGE` that has the advantage of reducing the session overhead spent preventing the delivery of duplicate messages.

Message Persistence

The default JMS delivery mode for a message is `PERSISTENT`. This ensures that the message will be delivered, even if the JMS provider fails or is shut down.

A second delivery mode, `NON_PERSISTENT`, can be used where guaranteed delivery is not required. A `NON_PERSISTENT` message has the lowest overhead because the JMS provider does not need to copy the message to a stable storage medium. JMS still guarantees to deliver a `NON_PERSISTENT` message at most once (but maybe not at all). Nonpersistent messages should be used when:

- Performance is important and reliability is not
- Messages can be lost with no effect on system functionality

Persistent and non-persistent messages can be delivered to the same destination.

Transactions

Often, acknowledgement of single messages is not enough to ensure the integrity of an application. Think of a banking system where two messages are sent to debit an amount from one account and credit the same amount to another. If only one of the messages is received, there will be a problem. A transaction is required where a number of operations involving many messages forms an atomic piece of work.

In JMS, you can specify that a session is transacted when a session queue or topic is created:

```
createQueueSession(boolean transacted, int acknowledgeMode)
createTopicSession(boolean transacted, int acknowledgeMode)
```

In a transacted session, several sends and receives are grouped together in a single transaction. The JMS API provides `Session.commit()` to acknowledge all the messages in a transaction and `Session.rollback()` to discard all messages. After a rollback, the messages will be redelivered unless they have expired.

To create a transacted queue session, set the transacted parameter to `true`, as shown in the following:

```
topicSession = topicConnection.createTopicSession(true, 0);
```

For transacted sessions, the `acknowledgeMode` parameter is ignored. The previous code sets this parameter to `0` to make this fact explicit.

There is no explicit transaction start. The contents of a transaction are simply those messages that have been produced and consumed during the current session, either since the session was created or since the last `commit()`. After a `commit()` or `rollback()`, a new transaction is started.

Because the `commit()` and `rollback()` methods are associated with a session, it is not possible to mix messages from queues and topics in the same transaction.

The following example shows a simple transaction involving two messages.

```
queueSession = queueConnection.createQueueSession(true, 0);
Queue bank1Queue = (Queue)context.lookup("queue/FirstUSA");
Queue bank2Queue = (Queue)context.lookup("queue/ArabBank");
bank1QueueSender = queueSession.createSender(bank1Queue);
bank2QueueSender = queueSession.createSender(bank2Queue);
// .. application processing to create debit and credit messages
try {
    bank1QueueSender.send(bank1Queue, debitMsg);
    bank2QueueSender.send(bank2Queue, creditMsg);
    queueSession.commit();
} catch(JMSException ex) {
    System.err.println("Exception in bank transaction:" + ex);
    queueSession.rollback();
}
```

Where a receiver handles atomic actions sent in multiple messages, it should similarly only commit when all the messages have been received and processed.

XA support

A JMS provider may provide support for distributed transaction using the X/Open XA resource interface. This is performed by utilizing the Java Transaction API (JTA). JTA was covered on Day 8, “Transactions and Persistence”. XA support is optional; refer to your JMS provider documentation to see if XA support is provided.

Multithreading

Not all the objects in JMS support concurrent use. The JMS API only specifies that the following objects can be shared by across multiple threads

- Connection Factories
- Connections
- Destinations

Many threads in the same client may share these objects, whereas the following:

- Sessions
- Message Producers
- Message Consumers

can only be accessed by one thread at a time. The restriction on single-threaded sessions reduces the complexity required by the JMS provider to support transactions.

Session concurrency can be implemented within a multithreaded client by creating multiple sessions.

Introduction to XML

Among the five message body types supported in the JMS API, the `TextMessage` type was included on the presumption that String messages will be used extensively. The reason for this presumption is the increasing use of the Extensible Markup Language (XML) for inter-application communication.

What Is XML and Why Would You Use It?

XML is a structured text-based language consisting of tags that are used to describe a document's structure and meaning. It does not say anything about the visual representation of the document.

XML is non-proprietary. It is also easy for both computers and people to understand. Consequently, it is an extremely useful format for the interchange of data between different applications. By defining not only the content but also the structure of the data, XML is ideally suited to manipulating arbitrary data structures.

There are many good reasons for using XML, but one of the most significant is its ability to adapt to changes in the data. Because of this, senders and receivers do not need to agree on a common data format ahead of time.

The following is an example of XML:

```
<person>
    <name>
        <firstname>Winston</firstname>
        <surname>Churchill</surname>
    </name>
    <birth>
        <date month='November' day='30' year='1874'</date>
        <place>Bleinham Palace, England</place>
    </birth>
    <death>
        <date> month='January' day='24' year='1965'</date>
        <buried>Bladon, Oxfordshire, </buried>
    </death>
</person>
```

Even with no knowledge of XML, it is easy to work out what this example is describing; and herein lies much of the power of XML.

XML will be covered in more detail on Day 16, " Integrating XML with J2EE" and an overview is provided in Appendix C, "An Overview of XML," on the CD-ROM.

Summary

Today, you have had an introduction to JMS messaging, the concept of message producers and consumers, and explored the two supported message domains—point-to-point and publish/subscribe. This has necessarily been an overview of JMS. More information should be obtained from the latest JMS specification and API. Also, refer to the documentation for your JMS provider to determine what features beyond those described here are supported.

You have also been given a brief introduction to XML and seen how it can be used to communicate with other applications.

Tomorrow, you will utilize your JMS knowledge gained today while examining the third type of EJB—Message-driven beans.

Q&A

Q What type of JMS message domain should be used to send a message to a single receiver?

A A point-to-point domain is the appropriate choice in this scenario.

Q What type of JMS message domain should be used to send a message to many receivers at the same time?

A To send to many receivers, the publish/subscribe message domain is the best choice.

Q What is the difference between `JMSHeader` fields and `JMSProperty` fields?

A JMS header fields are defined in the JMS API and are mainly set by the JMS provider. JMS property fields are used by clients to add additional header information to a message.

Q Does JMS guarantee to deliver a message in the point-to-point domain?

A Messages in the point-to-point domain are `PERSISTENT` by default and will be delivered unless they have a timeout that has expired. Point-to-point messages can be set to `NON_PERSISTENT`, in which case, the message may be lost if a provider fails.

Q When should I use a durable subscription?

A Durable subscriptions should be used when a subscriber needs to receive messages from a topic when it is inactive.

Exercise

To extend your knowledge of the subjects covered today, try the following exercises.

1. Create a chat room application. Participants provide their name and can send messages to any topic (hint: use a JMS property to define the topic). Participants may read messages posted by all other participants or filter by topic. You may use predefined topic names.

To assist you in this task, three Java files have been provided in the `exercise` sub-directory for Day 9 on the accompanying CD-ROM.

The `Chat.java` and `ChatDisplay.java` files are complete and need not be edited. These files provide the Swing code to enter and display the chat room messages onscreen.

The `TopicServer.java` is a starting point for you to further develop the chat server. The initial code simply uses the callback method `addMessage` to bounce the message back to the screen. The addMessage method uses the interface defined in `ChatDisplay.java`.

You will need to edit this file to replace this callback with code to publish the message to a topic. You then need to add a subscriber that consumes messages from this topic and displays them onscreen.

Add a property called `From` to the message and set it to the `from` parameter passed in. This will then be displayed in the chat room window.

A completed `TopicServer` is included in the `solutions` sub-directory of Day 9 of the case study.

WEEK 2

DAY 10

Message-Driven Beans

So far, you have looked at two types of Enterprise Java Bean (EJB)—the Session bean and the Entity bean. Today you will consider the third and final EJB, the Message-driven bean. Topics that are covered are as follows:

- Similarities and differences with Entity and Session beans
- The life-cycle of a Message-driven bean
- Writing a Message-driven bean

Prior to the EJB 2.0 specification, it was only possible to support asynchronous message passing by writing an external Java program that acted as a listener. A listener is a program whose sole purpose is to wait for data to arrive, for example, a socket server program that "listens" on a socket and perform some action when it detects client connections. The listener was then able to invoke methods on a session or entity bean. All EJB method calls had to be synchronous and initiated by the client. This approach had the disadvantage that the message was received outside of the server, so it could not be part of an EJB transaction.

With the release of J2EE 1.3, you can use Message-driven beans to combine the functionality of EJBs with the Java Message Service (JMS).

Although JMS was covered in detail on Day 9, "Java Message Service," the following is a quick recap of its main features:

- JMS is a Java API that specifies how applications can create, send, receive, and read messages.
- JMS enables communication that is both asynchronous and reliable, while minimizing the amount of knowledge and programming that is required.
- The implementation of the JMS API is provided by a number of vendors who are known as JMS providers.
- Message queues are associated with the point-to-point message domain. Messages in a queue are persistent but can only be consumed by one receiver.
- Topics allow a message to be sent to more then one receiver (called a subscriber). Messages are not persistent; they are immediately delivered to all existing subscribers.

What Are Message-Driven Beans?

Message-driven beans are generally constructed to be message consumers, although they can, like any other EJB, also be used to create and send messages. A Message-driven bean lives entirely within the container, it has no security context of its own. When the bean is deployed, it is associated with a particular queue or topic, and is invoked by the container when a message arrives for that queue or topic.

The following are the features of a Message-driven bean:

- It is anonymous; that is, it has no client visibility. No state is maintained for the client.
- All instances of a particular Message-driven bean are equivalent.
- The container can pool instances.
- It does not have a local or remote interface.
- It is invoked asynchronously by the container.
- The bean lives entirely within a container; the container manages its lifecycle and environment.

These features are discussed in more detail next.

The Message Producer's View

To the client producing JMS messages, the Message-driven bean is just an anonymous message consumer. The client need not be aware that the consumer is a Message-driven

bean. The client simply sends its messages to a destination, either a queue or a topic, and the bean handles the message when it arrives. Therefore, the coding of message producers in an application using Message-driven beans is exactly the same as any JMS application; that is, the message must conform to the JMS specification and the destination must be a Java Naming and Directory Interface (JNDI) registered name. Apart from this, the message does not have to correspond to any particular format.

It is not necessary for the client to be a Java client application or an EJB to take advantage of Message-driven beans; it can be a Java ServerPages™ (JSP) component or a non-J2EE application.

Similarities and Differences with Other EJBs

In some respects, a Message-driven bean is similar to a stateless Session bean. It is a complete EJB that can encapsulate business logic. An advantage is that the container is responsible for providing functionality for security, concurrency, transactions, and so forth. Like a Session or Entity bean, a Message-driven bean has a `bean` class and XML deployment descriptor.

The main difference from the other EJBs is that a Message-driven bean cannot be called directly by the client. For this reason, they do not have `Home`, `Remote`, or `Local` interfaces, this makes them less prone to misuse by the client.

Unlike Entity and Session beans, Message-driven beans do not have a passive state. Therefore, they do not implement the `ejbActivate()` and `ejbPassivate()` methods.

> Although a Message-driven bean is considered to be a stateless object, from the client's view, it can and should retain state in its instance variables. Examples of this are an open database connection and the `Home`, `Local`, and `Remote` interfaces to other EJBs.

Programming Interfaces in a Message-Driven Bean

There are a number of constraints on the contents of a Message-driven bean class. In particular your Message-driven bean class must

- Implement the `javax.ejb.MessageDrivenBean` interface
- Implement the `javax.jms.MessageListener` interface

- Have a single constructor, with no arguments
- Have a single public `setMessageDrivenContext(MessageDrivenContext ctx)` method that returns a void
- Have a single public `ejbCreate()` method with no arguments that returns a void
- Have a single public `ejbRemove()` method with no arguments that returns a void
- Have a single public `onMessage(Message message)` method that returns a void
- Not have a `finalize()` method

The following sections cover these methods in more detail.

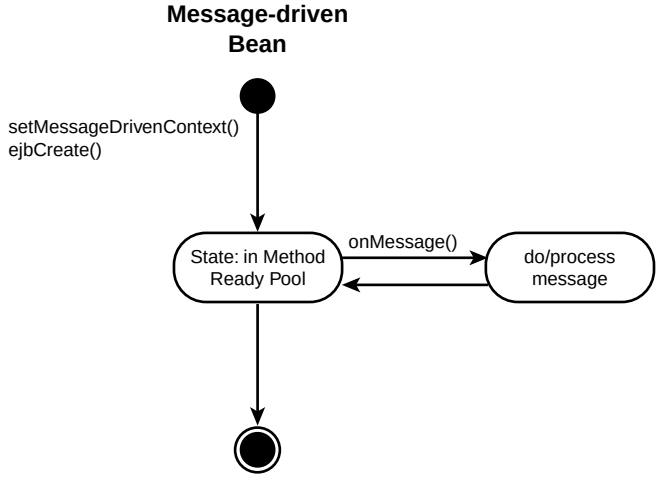
Life Cycle of a Message-Driven Bean

The EJB container controls the lifecycle of a Message-driven bean. The Message-driven bean instance lifecycle has three states, as shown in Figure 10.1:

- *Does Not Exist*—The Message-driven bean has not been instantiated yet or is awaiting garbage collection.
- *Method Ready Pool*—A pool of Message-driven bean instances, similar to the instance pool used for stateless session beans.
- *Processing a message*—The Message-driven bean has been instantiated and is handling a message.

FIGURE 10.1

The Message-driven bean life cycle.



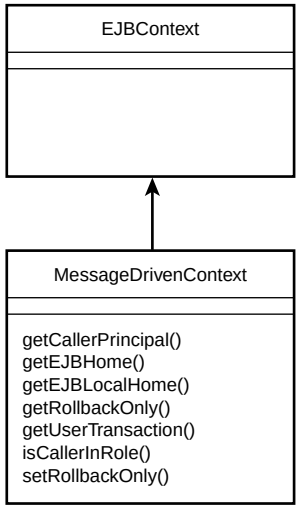
After constructing the new instance of the Message-driven bean object, the container invokes the following methods:

- The bean's `setMessageDrivenContext()` method with a reference to its EJB context. The Message-driven bean should store its `MessageDrivenContext` reference in an instance field.
- The bean's `ejbCreate()` method. The Message-driven bean's `ejbCreate()` method takes no arguments and is invoked only once when the bean is first instantiated.

The Message-Driven Bean Context

The `javax.ejb.MessageDrivenContext` interface (see the class diagram in Figure 10.2). provides the Message-driven bean with access to its runtime context. This is similar to the `SessionContext` and `EntityContext` interfaces for Session and Entity beans.

FIGURE 10.2
The `MessageDrivenContext` *class diagram.*



Note that all the `EJBContext` methods are available to a Message-driven bean, but because the Message-driven bean does not have a local or remote interface, calls to `getEJBHome()` and `getEJBLocalHome()` will throw a `java.lang.IllegalStateException`.

Because Message-driven beans are anonymous and run within the context of the container, and the container does not have a client security identity or role, calls to the `getCallerPrincipal()` and `IsCallerInRole()` methods will also cause an `IllegalStateException`.

Creating a Message-Driven Bean

The `setMessageDrivenContext()` method can throw `EJBException` if there is a container or system level error of some kind. See the section called "Handling Exceptions" for more details. What follows is an example setMessageDrivenContext() method that saves its `EJBContext` and JNDI context:

```
private MessageDrivenContext mdbContext;
private Context jndiContext;
public void setMessageDrivenContext (MessageDrivenContext ctx) {
    mdbContext = ctx;
    try { jndiContext = new InitialContext();
    } catch (NamingException nameEx) {
        throw new EJBException(nameEx);
    }
}
```

After calling `setMessageDrivenContext()`, the container calls the bean's `ejbCreate()` method, which takes no parameters. You could use this method to allocate resources, such as a datasource, but in practice, this is usually done in the `setMessageDrivenContext()` method. Therefore, it is normal to find the `ejbCreate()` method empty.

This method is only invoked when the bean instance is first created.

```
public void ejbCreate () throws CreateException
```

After the `ejbCreate()` method has been called, the bean is placed in the method-ready pool.

Method-Ready Pool

The actual point at which Message-driven bean instances are created and placed in the method-ready pool is vendor specific. The vendor of an EJB server could design it to only create Message-driven bean instances when they are required. Alternatively, when the EJB server is started, a number of instances may be placed in the method-ready pool awaiting the first message. Additional instances can be added to the pool when the number of Message-driven beans is insufficient to handle the number of incoming messages.

Therefore, the life of a Message-driven bean instance could be very long and, in this case, it makes sense to adopt an approach where you retain state (such as an open database connection) across the handling of several messages. However, the container may create and destroy instances to service every incoming message. If this is the case, this approach is no longer efficient. Check your vendor's documentation for details on how your EJB server handles Message-driven bean instances in the method-ready pool.

Message-driven bean instances in the method-ready pool are available to consume incoming messages. Any available instance can be allocated to a message and, while processing the message, this particular bean instance is not available to consume other messages. A container can handle several messages concurrently by using a separate instance of the message bean for each message. Each separate instance obtains its own `MessageDrivenContext` from the container. After the message has been processed, the instance is available to consume other messages. Message-driven beans are always single-threaded objects.

The Demise of the Bean

When the server decides to reduce the total size of the method-ready pool, a bean instance is removed from the pool and becomes available for garbage collection. At this point, the bean's `ejbRemove()` method is called.

You should use this method to close or deallocate resources stored in instance variables and set the instance variable to null.

```
public void ejbRemove()
```

The `EJBException` can be thrown by `ejbRemove()` to indicate a system-level error.

Following `ejbRemove()`, the bean is dereferenced and no longer available to handle messages. It will eventually be garbage collected.

The `ejbRemove()` method may not be called if the Message-driven bean instance throws an exception. This could result in resource leaks.

A Message-driven bean must not define the finalize method to free up resources: do all the tidying up in `ejbRemove()`.

Consuming Messages

When a message is received, the container finds a Message-driven bean instance that is registered for that queue or topic and calls the bean's `onMessage()` method.

```
public void onMessage(Message message)
```

This method has a single parameter that contains a single JMS message. The message will have a header, one or more properties (optional), and a message body (consisting of one of the five JMS message body types). JMS messages were covered in some detail on Day 9.

The Message-driven bean must provide a single `onMessage()` method, and this method should not throw runtime exceptions. It must not have a `throws` clause as part of its method signature. The `onMessage()` holds the business logic of the bean. You can use helper methods and other EJBs to process the message.

Remember, Message-driven bean instances are triggered asynchronously; the business logic within the bean must reflect this. You must never presume any ordering to the messages received. Even if the system is implemented within the same JVM, the system vagaries can cause the scheduling of bean instances to be non-deterministic, this means that you cannot ascertain or control when the bean will run.

Handling Exceptions

The Message-driven bean can encounter various exceptions or errors that prevent it from successfully completing. The following are examples of such exceptions:

- Failure to obtain a database connection
- A JNDI naming exception
- A `RemoteException` from invocation of another EJB
- An unexpected `RuntimeException`

A well-written Message-driven bean should never carelessly throw a `RunTimeException`. If a `RunTimeException` is not caught in `onMessage()` or any other bean class method, the container will simply discard the instance (it will transition it to the Does Not Exist state). In this case, the container will not call the `ejbRemove()` method, so a badly written bean method could cause resource leaks.

Obviously, you need a mechanism to tell the container that you have caught an unrecoverable error and die gracefully. To do this, you use exception layering. You catch the `RunTimeException`, free up resources, do any other appropriate processing and then throw an `EJBException` to the container. The container will then log the error, rollback any container-managed transactions, and discard the instance.

Because identical Message-driven bean instances are available, from the client perspective, the message bean continues to exist in the method-ready pool to handle further messages. Therefore, a single instance failure may not cause any disruption to the system.

Container- and Bean-Managed Transactions

The analysis of container- versus bean-managed transactions was covered as part of Day 8's material, reread this if you need to recap the benefits of either method of handling transactions. When designing your Message-driven bean, you must decide whether the

bean will demarcate the transactions programmatically (bean managed transactions), or if the transaction management is to be performed by the container. This is done by setting the `transaction-type` in the deployment descriptor.

```
<transaction-type>Container</transaction-type>
```

Use container-managed transactions unless you have some other reason for using bean-managed transactions, such as creating and sending a series of messages.

A Message-driven bean can be designed with either bean-managed transactions or with container-managed transactions, but both cannot be used in the same bean.

The following methods in the `javax.ejb.MessageDrivenContext` (all inherited from `javax.ejb.EJBContext`) can be used with transactions.

```
public UserTransaction getUserTransaction() throws
java.lang.IllegalStateException
```


The `UserTransaction` interface methods can be used to demarcate transactions.

The `getUserTransaction()` method can only be called if the Message-driven bean is using bean-managed transactions. An attempt to use this method from a bean using container-managed transactions will cause a `java.lang.IllegalStateException` to be thrown.

Both the methods `setRollbackOnly()` and `getRollbackOnly()` can only be used with container-managed transactions. This time, the `IllegalStateException` will be thrown if they are utilized in the context of a bean-managed transaction.

```
public void setRollbackOnly() throws java.lang.IllegalStateException
```

Typically, you use `setRollbackOnly()` after an exception or error of some kind to mark the current transaction to be rolled back.

```
public boolean getRollbackOnly() throws java.lang.IllegalStateException
```

The `getRollbackOnly()` method returns `true` if the transaction has been marked for rollback; otherwise, it returns `false`. You usually call this method after an exception has been caught to see if there is any point in continuing working on the current transaction.

Message Acknowledgment

With Message-driven beans, the container handles message acknowledgement. The default being `AUTO_ACKNOWLEDGE`.

If you use container-managed transactions, you have no control over the message acknowledgement; it is done automatically as part of the transaction commit.

With bean-managed transactions, you can specify `DUPS_OK_ACKNOWLEDGE` as an alternative to the default. To do this, set the `acknowledge-mode` element in the deployment descriptor. With `DUPS_OK_ACKNOWLEDGE` set, you can reduce the session overhead spent preventing delivery of duplicate messages, but only do this if receiving duplicate messages will not cause a problem with the business logic of your bean.

```
<transaction-type>Bean</transaction-type>
<acknowledge-mode>Dups-ok-acknowledge</acknowledge-mode>
```

JMS Message Selectors

JMS message selectors was covered in detail on Day 9. A message selector is a string containing an SQL conditional expression. Only JMS message header values and message properties can be specified in the message selector.

With Message-driven beans, the selector is specified at deployment time.

The message selector is added to the screen in the Deployment Tool (see Figure 10.3). In the example shown, the bean will handle only messages that have a `JMSPriority` greater than the default of `4`.

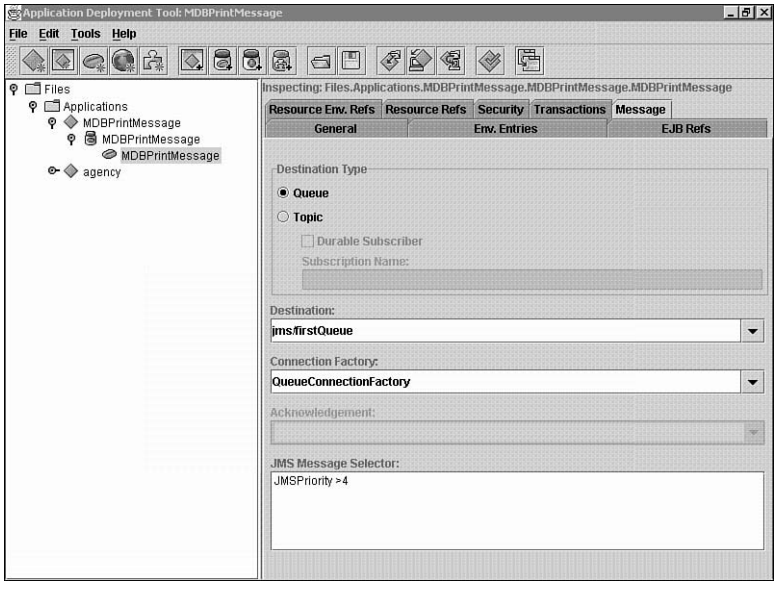


FIGURE 10.3

Deployment Tool screen showing the setting of message selectors.

The deployment descriptor is updated to include the message-selector tag.

```
<message-selector>JMSPriority &gt;4</message-selector>
```

Writing a Simple Message-Driven Bean

As you work through this section, you will create a Message-driven bean that that simply prints out the contents of a text message on screen.

So that the Message-driven bean can work asynchronously you will employ the `MessageListener` interface. This interface and the associated `onMessage()` method, which is invoked each time a message is available at the destination, were fully described in Day 9.

Implementing the Interfaces

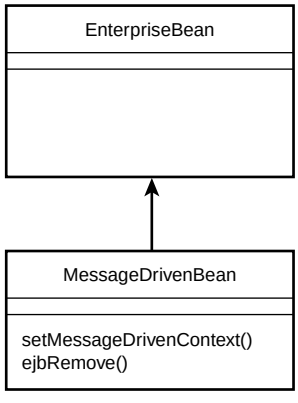
As already stated, all Message-driven beans must implement the `MessageDrivenBean` and `MessageListener` interfaces.


```
import javax.ejb.*;
import javax.jms.*;
public class MDBPrintMessage implements MessageDrivenBean, MessageListener {
// class body not shown – see listing 10.1
}
```

Just like the `EntityBean` and `SessionBean` interfaces, the `MessageDrivenBean` interface extends the `javax.ejb.EnterpriseBean` interface.

The `MessageDrivenBean` interface contains only two methods—`setMessageDrivenContext()` and `ejbRemove()`, see the class diagram in Figure 10.4.

FIGURE 10.4
The `MessageDrivenBean` *class diagram.*



You also need to supply an `ejbCreate()` method.

In this example, we have no need to create or store resources, and it is so simple that we will leave all the required methods blank.

```
public void setMessageDrivenContext (MessageDrivenContext ctx) {}
public void ejbRemove() {}
public void ejbCreate() {}
```

The `MessageListener` interface is where the Message-driven bean carries out the bean's business logic. As already stated, it consists of the single method `onMessage()`. In this example, we will simply test that the message is a `TextMessage` and, if it is, print it to the screen. The full code is shown in Listing 10.1.

LISTING 10.1 Simple Print Message Message-Driven Bean

```
1: import javax.ejb.*;
 2: import javax.jms.*;
 3:
 4: public class MDBPrintMessage implements MessageDrivenBean, MessageListener {
 5:
 6:     public void setMessageDrivenContext (MessageDrivenContext ctx) {}
 7:     public void ejbRemove() {}
 8:     public void ejbCreate() {}
 9:
10:     public void onMessage(Message message) {
11:         try {
12:             if (message instanceof TextMessage)
13:             {
14:                  String text = ((TextMessage) message).getText();
15:                  System.out.println("Received: " + text);
16:             }
17:         } catch(Exception ex) {
18:             throw new EJBException(ex);
19:         }
20:     }
21: }
```

As you can see, there is no reference in this code to any particular JMS queue or topic. This means that the bean is not only generic and can be associated with any queue or topic at deployment time, it can also be associated with several different queues or topics at the same time.

Running the Example

Before you can see your Message-driven bean working, there are a number of steps still to go through:

1. Compile the bean.
2. Use `j2eeadmin` or `deploytool` to create the message queue.

3. Deploy the bean.
4. Create a sender client to create a message.

Creating the Queue

The message bean is associated with a queue or topic at deployment time. This queue must already exist. To create a queue (or topic), do the following:

1. Ensure that J2EE server is running.
2. Use `j2eeadmin` or deploytool to create the message queue or a topic.

   To see the existing queues and topics, use the following:

   ```
   j2eeadmin -listJMSDestination
   ```

   or view the Destinations screen in `deploytool`. This is found by selecting Server Configuration from the Tools menu and then the Destinations icon in the left panel.

The J2EE RI has two default queues predefined—`jms/Queue` and `jms/Topic`.

To add your queue, use the following:

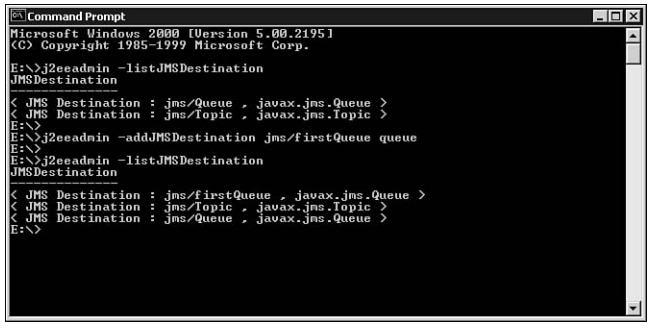
```
j2eeadmin -addJMSDestination jms/firstQueue queue
```

The `j2eeadmin` command works silently, so to check that your queue has been created, run `J2eeadmin -listJMSDestination` once more. Figure 10.5 demonstrates the use of these two commands to create a queue called `jms/firstQueue` (this is the queue you will use in this first example).

Alternatively, you can add the queue in `deploytool` on the Destinations screen.

FIGURE 10.5

Using `j2eeadmin` *to add a JMS queue to the container.*



When your bean is deployed, the following will appear in the XML deployment descriptor.

```
<message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
</message-driven-destination>
```

Deploying the Message-Driven Bean

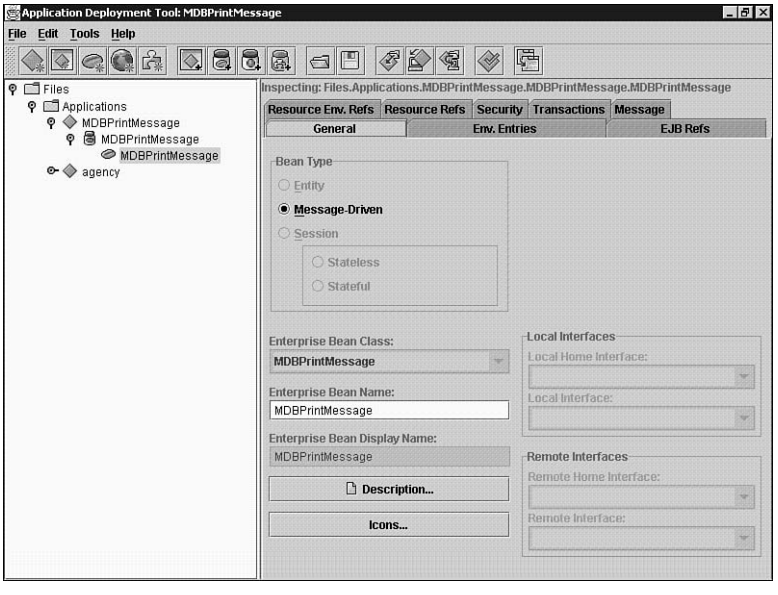
By this time, you should be familiar with deploying Entity and Session beans. This section will only cover in any detail where the process differs for Message-driven beans.

The steps are as follows:

1. Run the deploytool.
2. Create a new application to hold your bean called **`MDBPrintMessage`**.
3. Select New Enterprise Bean from the File menu and add the `MDBPrintMessage.class` file to the `MDBPrintMessage` application JAR file.
4. On the next screen, where you choose the type of enterprise bean that you are creating, select the bean type to be Message-Driven—most of the screen will blank out at this point.
5. Select `MDBPrintMessage` from the drop-down list for Enterprise Bean Class. The Enterprise Bean Name will be filled in automatically (see Figure 10.6).

FIGURE 10.6

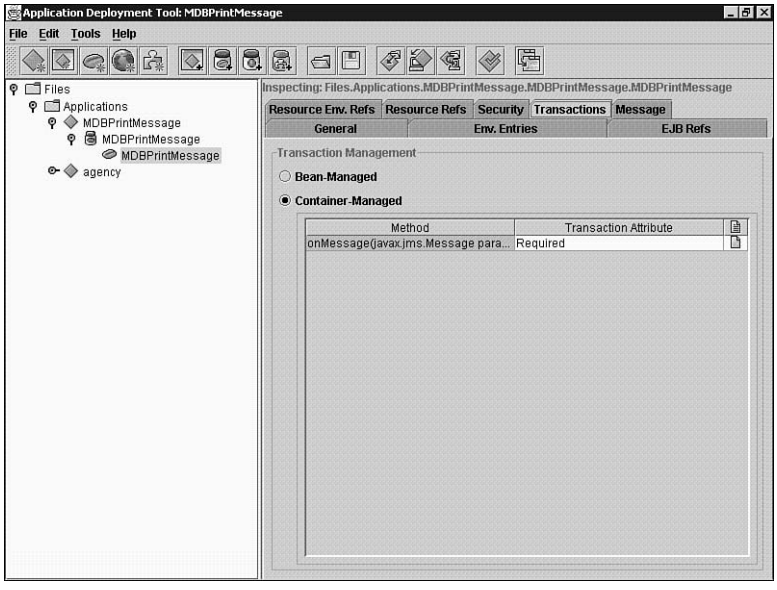
Selecting the Message-Driven bean.



6. On the next Transaction Management screen, select Container-Managed (see Figure 10.7).

FIGURE 10.7

Selecting Container-Managed transactions.



7. On the Message-Driven Bean Settings, select:

   Destination type: queue

   Destination: `jms/firstQueue`

   Connection Factory: `QueueConnectionFactory`

   Leave the JMS Message Selector blank (see Figure 10.8).

8. Select Finish.
9. Select the Verifier from the Tools menu. If the Verifier indicates that the bean has failed a `tests.ejb.SecurityIdentityRefs` test, this can safely be ignored. Security roles do not apply to a Message-driven bean because it runs within the identity of the container.
10. Select Deploy from the Tools menu to bring up the screen shown in Figure 10.9.
11. You will not need the Client Jar, so deselect it.
12. Select Finish and check that the bean has been successfully deployed.

Listing 10.2 shows the XML deployment descriptor that has been created for you. Highlighted in bold are those items that are of interest to you as a Message-driven bean writer.

FIGURE 10.8

Selecting JNDI references for Destination and Connection Factory.

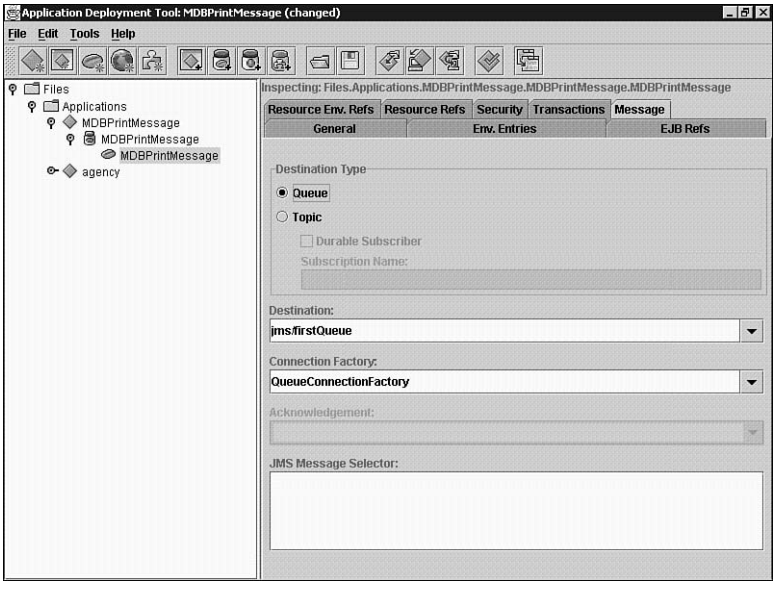
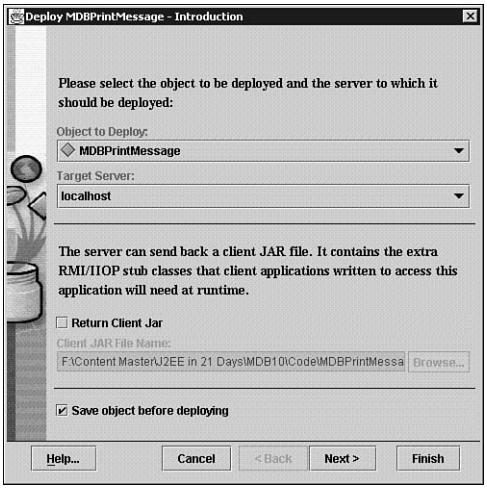


FIGURE 10.9

Deploying the `MDBPrintMessage` *bean.*


Deploy MDBPrintMessage - Introduction
Please select the object to be deployed and the server to which it should be deployed:
Object to Deploy:
MDBPrintMessage
Target Server:
localhost
The server can send back a client JAR file. It contains the extra RMI/IIOP stub classes that client applications written to access this application will need at runtime.
Return Client Jar
Client JAR File Name:
F:\Content Master\J2EE in 21 Days\MDB10\Code\MDBPrintMessa
Browse...
Save object before deploying
Help...
Cancel
< Back
Next >
Finish

LISTING 10.2 Deployment Descriptor for MDBPrintMessage

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans
 3: ➥ 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
 4: <ejb-jar>
 5:   <display-name>MDBPrintMessage</display-name>
```

LISTING 10.2 Continued

```
 6:   <enterprise-beans>
 7:     <message-driven>
 8:       <display-name>MDBPrintMessage</display-name>
 9:       <ejb-name>MDBPrintMessage</ejb-name>
10:       <ejb-class>MDBPrintMessage</ejb-class>
11:       <transaction-type>Container</transaction-type>
12:       <message-driven-destination>
13:         <destination-type>javax.jms.Queue</destination-type>
14:       </message-driven-destination>
15:     </message-driven>
16:   </enterprise-beans>
17:   <assembly-descriptor>
18:     <container-transaction>
19:       <method>
20:         <ejb-name>MDBPrintMessage</ejb-name>
21:         <method-intf>Bean</method-intf>
22:         <method-name>onMessage</method-name>
23:         <method-params>
24:           <method-param>javax.jms.Message</method-param>
25:         </method-params>
26:       </method>
27:       <trans-attribute>Required</trans-attribute>
28:     </container-transaction>
29:   </assembly-descriptor>
30: </ejb-jar>
```


Create a Sender Client to Create a Message

So far, you have created a Message-driven bean that is (as far as you are concerned) waiting to handle any message sent to the `jms/firstQueue` queue. All that is left to do is send a message to that queue and check that your bean is working correctly.

You can use the code or the `PTPSender` program described in Day 9 to send the message. This is not an EJB, it is a simple client application, so it does not need to be deployed. This code has been reproduced in Listing 10.3. for completeness.

LISTING 10.3 Point-to-Point Sender Code to Create and Send a Message to the `jms/firstQueue` Queue

```
1: import javax.naming.*;
2: import javax.jms.*;
3:
4: public class PTPSender {
5:
6:     private QueueConnection queueConnection;
7:     private QueueSession queueSession;
```

LISTING 10.3 Continued

```
 8:     private QueueSender queueSender;
 9:     private Queue queue;
10:
11:     private static final String jndiFactory = "QueueConnectionFactory";
12:     private static final String jndiQueue = "jms/firstQueue";
13:
14:     public static void main(String[] args) {
15:     try {
16:
17:         PTPSender sender = new PTPSender(jndiFactory, jndiQueue);
18:         System.out.println ("Sending message to jms/firstQueue");
19:         sender.sendMessage("Here is a message sent to jms/firstQueue");
20:          sender.close();
21:     } catch(Exception ex) {
22:          System.err.println("Exception in PTPSender: " + ex);
23:     }
24: }
25:
26:     public PTPSender(String jndiFactory, String jndiQueue)
27: ➥ throws JMSException, NamingException {
28:         Context context = new InitialContext();
29:         QueueConnectionFactory queueFactory =
30: ➥ (QueueConnectionFactory)context.lookup(jndiFactory);
31:         queueConnection = queueFactory.createQueueConnection();
32:         queueSession = queueConnection.createQueueSession(false,
33: ➥ Session.AUTO_ACKNOWLEDGE);
34:         queue = (Queue)context.lookup(jndiQueue);
35:         queueSender = queueSession.createSender(queue);
36:     }
37:
38:     public void sendMessage(String msg) throws JMSException {
39:         TextMessage message = queueSession.createTextMessage();
40:         message.setText(msg);
41:         queueSender.send(message);
42:         }
43:
44:     public void close() throws JMSException {
45:          //Send a non-text control message indicating end of messages
46:         queueSender.send(queueSession.createMessage());
47:
48:         queueSender.close();
49:         queueSession.close();
50:         queueConnection.close();
51:     }
52: }
```

Run the `PTPSender` program from the command line to put a message in the queue `jms/firstQueue`.

Check that you see the message:

```
"Here is a message sent to jms/firstQueue"
```

Now check that your message bean has received the message. If you started the J2EE RI with the `-verbose` switch, you will see the output of the Message-driven bean in the server window. If not, the output will be in the server log file.

```
"Received: Here is a message sent to jms/firstQueue "
```

Developing the Agency Case Study Example

Now you will turn your attention to a more realistic example. You will extend the Agency case study to utilize a Message-driven bean to match advertised jobs to new applicants as they register with the system or when an applicant updates his or her skills or location.

The steps are as follows:

1. Write a helper class that creates and sends a message to the `jms/applicantQueue` containing the applicant's login.
2. Amend the `Agency` and `Register` Session beans to call this new method when a new applicant is registered or the applicant's location or skills are changed.
3. Write a Message-driven bean to
    - Consume a message on the `jms/applicantQueue`
    - Look up the applicant's location and skills information
    - Find all the jobs that match the applicant's location
    - For each of these jobs, find those that require the applicant's skills
    - Determine if the applicant has all or just some of the skills
    - Store applicant and job matches in the Matched table
4. Create the `jms/applicantQueue` queue.
5. Deploy the new EJBS; run and test the application.

Step 1—Sender Helper Class

This class contains a constructor for the class and two methods—`sendApplicant()` and `close()`.

The constructor takes two parameters, which are strings representing the JNDI names of the JMS connection factory and the JMS queue.

```
public MessageSender(String jndiFactory, String jndiQueue)
➥ throws JMSException, NamingException {
```

```
    Context context = new InitialContext();
QueueConnectionFactory queueFactory =
➥ (QueueConnectionFactory)context.lookup(jndiFactory);
    queueConnection = queueFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false,
➥ Session.AUTO_ACKNOWLEDGE);
    queue = (Queue)context.lookup(jndiQueue);
    queueSender = queueSession.createSender(queue);
}
```

The `sendApplicant()` method is called by the `Agency` Session bean when a new applicant registers with the system and the `Register` Session bean when an existing applicant changes his or her location or job skills. It has two parameters—the applicant's login string and a Boolean denoting if this is a new applicant.

The `close()` method is called before the application is terminated. It sends a message that lets the container know that no more messages will be sent to the queue and frees-up resources.

The code for the MessageSender in shown in Listing 10.4, it should be very familiar by now.

LISTING 10.4 `MessageSender` Helper Class

```
 1: import javax.naming.*;
 2: import javax.jms.*;
 3:
 4: public class MessageSender {
 5:
 6:     private QueueConnection queueConnection;
 7:     private QueueSession queueSession;
 8:     private QueueSender queueSender;
 9:     private Queue queue;
10:
11:     public MessageSender(String jndiFactory, String jndiQueue)
12: ➥ throws JMSException, NamingException {
13:         Context context = new InitialContext();
14:         QueueConnectionFactory queueFactory =
15: ➥ (QueueConnectionFactory)context.lookup(jndiFactory);
16:         queueConnection = queueFactory.createQueueConnection();
17:         queueSession = queueConnection.createQueueSession(false,
18: ➥ Session.AUTO_ACKNOWLEDGE);
19:         queue = (Queue)context.lookup(jndiQueue);
20:         queueSender = queueSession.createSender(queue);
21:     }
22:
23:     public void sendApplicant (String applicant, boolean newApplicant)
24: ➥ throws JMSException {
```

LISTING 10.4 Continued

```
25:         TextMessage message = queueSession.createTextMessage();
26:         message.setBooleanProperty ("NewApplicant", newApplicant);
27:         message.setText(applicant);
28:         queueSender.send(message);
29:         }
30:
31:     public void close() throws JMSException {
32:          //Send a non-text control message indicating end of messages
33:         queueSender.send(queueSession.createMessage());
34:
35:         queueSender.close();
36:         queueSession.close();
37:         queueConnection.close();
38:     }
39: }
```


Step 2—Agency and Register Session Bean

The following changes are required to `AgencyBean.java` and `RegisterBean.java` to call the `MessageSender.send()` method when a new applicant is registered or the applicant's location or skills are changed.

1. In both `AgencyBean.java` and `RegisterBean.java` create a `MessageSender` object in the `setSessionContext()` method..

```
private MessageSender messageSender;
public void setSessionContext(SessionContext ctx) {
    /* existing code */
messageSender = new MessageSender (
➥ "java:comp/env/jms/QueueConnectionFactory",
➥ "java:comp/env/jms/applicantQueue");
}
```

2. In the `AgencyBean.java` file, add code to send a message indicating that a new applicant has registered in the `createApplicant()` method. The added line is shown in bold in the following code.

```
public void createApplicant(String login, String name,
➥ String email) throws DuplicateException, CreateException{
    try {
        ApplicantLocal applicant =
➥ applicantHome.create(login,name,email);
        messageSender.sendApplicant(applicant.getLogin(),true);
    }
    catch (CreateException e) {
        error("Error adding applicant "+login,e);
    }
```

```
    catch (JMSException e) {
        error("Error sending applicant details to message
➥ bean "+login,e);
    }
```

3. In the `RegisterBean.java` file, change `updateDetails()` to send a message to indicate that the applicant's details have changed. The added lines are shown in bold in the following code.

```
public void updateDetails (String name, String email,
➥ String locationName, String summary, String[] skillNames) {

    List skillList;
    try {
        skillList = skillHome.lookup(Arrays.asList(skillNames));
    } catch(FinderException ex) {
        error("Invalid skill", ex); // throws an exception
        return;
    }

    LocationLocal location = null;
    if (locationName != null) {
        try {
            location =
➥ locationHome.findByPrimaryKey(locationName);
        } catch(FinderException ex) {
            error("Invalid location", ex);
            return;
        }
    }
    applicant.setName(name);
    applicant.setEmail(email);
    applicant.setLocation(location);
    applicant.setSummary(summary);
    applicant.setSkills( skillList );
    try {
        messageSender.sendApplicant(applicant.getLogin(),false);
    }
    catch (JMSException ex) {
        ctx.setRollbackOnly();
        error ("Error sending applicant match message",ex);
    }
}
```

4. In both `AgencyBean.java` and `RegisterBean.java`, add the following to `ejbRemove()` to close down the `MessageSender`.

```
try {
        messageSender.close();
    }
    catch (JMSException ex) {
```

```
            error("Error closing down the queue",ex);
        }
```

5. Compile and deploy this code.

Step 3—The Message-Driven Bean

Although this Message-driven bean is significantly larger than your previous example, it does essentially the same thing.

This time, you need to obtain the JNDI `InitialContext` and use it to obtain references to various Entity beans used in the code.

```
public void setMessageDrivenContext(MessageDrivenContext ctx) {
    InitialContext ic = null;
        try {
            ic = new InitialContext();
            applicantHome = (ApplicantLocalHome)ic.lookup(
➥ "java:comp/env/ejb/ApplicantLocal");
        }
        catch (NamingException ex) {
            error("Error connecting to java:comp/env/ejb/ApplicantLocal:",ex);
        }
        try {
            jobHome = (JobLocalHome)ic.lookup("java:comp/env/ejb/JobLocal");
        }
        catch (NamingException ex) {
            error("Error connecting to java:comp/env/ejb/JobLocal:",ex);
        }
        try {
            matchedHome = (MatchedLocalHome)ic.lookup(
➥ "java:comp/env/ejb/MatchedLocal");
        }
        catch (NamingException ex) {
            error("Error connecting to java:comp/env/ejb/MatchedLocal:",ex);
        }
    }
```

The `ejbCreate()` method is blank.

```
public void ejbCreate(){}
```

The `ejbRemove()` cleans up by setting all the references to the Entity beans to `null`. There are no other resources for you to deallocate.

```
public void ejbRemove(){
    applicantHome = null;
    jobHome = null;
    matchedHome = null;
}
```

The `onMessage()` method is where you will code the algorithm that matches an applicant to advertised jobs. First, you check that `onMessage()` has received the expected text message. The message contains the applicant's login, which is the primary key on the `Applicants` table. This primary key is used to obtain the applicant's location and skills in subsequent finder methods.

```
String login = null;
try {
    if (!(message instanceof TextMessage)) {
System.out.println("ApplicantMatch: bad message:" + message.getClass());
        return;
    }
```

Now, check if this applicant is a new one or if he or she has amended his or her registration. If the applicant has changed his or her details, you need to delete the existing matches stored in the `Matched` table.

```
try {
    login = ((TextMessage)message).getText();
    if (! message.getBooleanProperty("NewApplicant")) {
        matchedHome.deleteByApplicant(login);
    }
}
catch (JMSException ex) {
    error ("Error getting JMS property: NewApplicant",ex);
}
```

Use the login primary key to find the applicant's location using the `Applicant` Entity bean's finder method.

```
try {
    ApplicantLocal applicant = applicantHome.findByPrimaryKey(login);
    String location = applicant.getLocation().getName();
```

Next, obtain all the skills that the applicant has registered and store them in an array.

```
    Collection skills = applicant.getSkills();
    Collection appSkills = new ArrayList();
    Iterator appIt = skills.iterator();
    while (appIt.hasNext()) {
        SkillLocal as = (SkillLocal)appIt.next();
        appSkills.add(as.getName());
    }
```

Now you have all the information you need about the applicant. The next step is to start matching the jobs. First find the jobs that match the applicant's location from the `Job` bean.

```
    Collection col = jobHome.findByLocation(location);
```

Iterate over this collection finding the skills required for each job.

```
    Iterator jobsIter = col.iterator();
    while (jobsIter.hasNext()) {
        JobLocal job = (JobLocal)jobsIter.next();
        Collection jobSkills = job.getSkills();
```

Now you have a `appSkills` array containing the applicant's skills and a `jobSkills` collection containing the skills required for the job. The next task is to find how many of these skills match. This is done by iterating over the `jobSkills`, and for each `jobSkill`, searching the `appSkills` array for a match. When a match is found, the `skillMatch` counter is incremented.

```
    int skillMatch = 0;
        Iterator jobSkillIter = jobSkills.iterator();
        while (jobSkillIter.hasNext()) {
            SkillLocal jobSkill = (SkillLocal)jobSkillIter.next();
            if (appSkills.contains(jobSkill.getName()))
              skillMatch++;
        }
```


Now see if you have a match. If there was a job skill to match (`jobSkills.size() >0`) and the applicant did not have any of them (`skillMatch == 0`), get the next job (`continue`).

```
    if (jobSkills.size() > 0 && skillMatch == 0)
        continue;
```

Otherwise, determine if the applicant has all or just some of the skills.

```
    boolean exact = skillMatch == jobSkills.size();
```

You are now in a position to update the `Matched` table. First create the primary key for this table.

```
    MatchedPK key = new MatchedPK(login,job.getRef(),job.getCustomer());
```

Now all that is left to do is store the applicant and job details in the `Matched` table.

```
    try {
matchedHome.create(key.getApplicant(), key.getJob(),
➥ key.getCustomer(), exact);
    }
catch (CreateException ex) {System.out.println(
➥ "ApplicantMatch: failed to create matched entry: "+key);
    }
```

That is all there is to the bean apart from the exception handling. The full listing of the `ApplicantMatch` Message-driven bean is shown in Listing 10.5.

LISTING 10.5 Full Listing on `ApplicantMatch.java` Message-Driven Bean Code

```
 1: package data;
 2:
 3: import java.util.*;
 4: import javax.ejb.*;
 5: import javax.jms.*;
 6: import javax.naming.*;
 7:
 8: public class ApplicantMatch implements MessageDrivenBean, MessageListener
 9: {
10:     private ApplicantLocalHome applicantHome;
11:     private JobLocalHome jobHome;
12:     private MatchedLocalHome matchedHome;
13:
14:     public void onMessage(Message message) {
15:         String login = null;
16:         if (!(message instanceof TextMessage)) {
17:             System.out.println("ApplicantMatch: bad message:" +
18: ➥ message.getClass());
19:             return;
20:         }
21:         try {
22:             login = ((TextMessage)message).getText();
23:             if (! message.getBooleanProperty("NewApplicant")) {
24:                 matchedHome.deleteByApplicant(login);
25:             }
26:         }
27:         catch (JMSException ex) {
28:             error ("Error getting JMS property: NewApplicant",ex);
29:         }
30:         try {
31:             ApplicantLocal applicant =
➥applicantHome.findByPrimaryKey(login);
32:             String location = applicant.getLocation().getName();
33:             Collection skills = applicant.getSkills();
34:             Collection appSkills = new ArrayList();
35:             Iterator appIt = skills.iterator();
36:             while (appIt.hasNext()) {
37:                 SkillLocal as = (SkillLocal)appIt.next();
38:                 appSkills.add(as.getName());
39:             }
40:             Collection col = jobHome.findByLocation(location);
41:             Iterator jobsIter = col.iterator();
42:             while (jobsIter.hasNext()) {
43:                 JobLocal job = (JobLocal)jobsIter.next();
44:                 Collection jobSkills = job.getSkills();
45:                 int skillMatch = 0;
46:                 Iterator jobSkillIter = jobSkills.iterator();
47:                 while (jobSkillIter.hasNext()) {
```

LISTING 10.5 Continued

```
48:                     SkillLocal jobSkill = (SkillLocal)jobSkillIter.next();
49:                     if (appSkills.contains(jobSkill.getName()))
50:                        skillMatch++;
51:                 }
52:                 if (jobSkills.size() > 0 && skillMatch == 0)
53:                     continue;
54:                 boolean exact = skillMatch == jobSkills.size();
55:                 MatchedPK key = new MatchedPK(login,job.getRef(),
56: ➥ job.getCustomer());
57:                 try {
58:                     matchedHome.create(key.getApplicant(),key.getJob(),
59: ➥ key.getCustomer(), exact);
60:                 }
61:                 catch (CreateException ex) {
62:                     System.out.println("ApplicantMatch: failed to create
63: ➥ matched entry: "+key);
64:                 }
65:             }
66:         }
67:         catch (FinderException ex) {
68:             System.out.println("ApplicantMatch: failed to find applicant
69: ➥ data: "+login);
70:         }
71:         catch (RuntimeException ex) {
72:             System.out.println("ApplicantMatch: "+ex);
73:             ex.printStackTrace();
74:             throw ex;
75:         }
76:     }
77:
78:     // EJB methods start here
79:
80:     public void setMessageDrivenContext(MessageDrivenContext ctx) {
81:     InitialContext ic = null;
82:         try {
83:             ic = new InitialContext();
84:             applicantHome = (ApplicantLocalHome)ic.lookup(
85: ➥ "java:comp/env/ejb/ApplicantLocal");
86:         }
87:         catch (NamingException ex) {
88:             error("Error connecting to
➥java:comp/env/ejb/ApplicantLocal:",ex);
89:         }
90:         try {
91:             jobHome =
➥(JobLocalHome)ic.lookup("java:comp/env/ejb/JobLocal");
92:         }
93:         catch (NamingException ex) {
```

LISTING 10.5 Continued

```
 94:             error("Error connecting to java:comp/env/ejb/JobLocal:",ex);
 95:         }
 96:         try {
 97:             matchedHome = (MatchedLocalHome)ic.lookup(
 98: ➥ "java:comp/env/ejb/MatchedLocal");
 99:         }
100:         catch (NamingException ex) {
101:             error("Error connecting to
➥java:comp/env/ejb/MatchedLocal:",ex);
102:         }
103:     }
104:
105:     public void ejbCreate(){
106:     }
107:
108:     public void ejbRemove(){
109:         applicantHome = null;
110:         jobHome = null;
111:         matchedHome = null;
112:     }
113:
114:     private void error (String msg, Exception ex) {
115:         String s = "ApplicantMatch: "+msg + "\n" + ex;
116:         System.out.println(s);
117:         throw new EJBException(s,ex);
118:     }
```

Compile this bean.

Step 4—Create the JMS Queue

Run the J2EE RI and use `j2eeadmin` to create the JMS queue.

```
j2eeadmin -addJMSDestination jms/applicantQueue queue
```

Alternatively, use `deploytool` and select Destinations from the Configure Installation screen and add the queue.

Step 5—Deploy the EJBS

Now deploy the `ApplicantMatch` Message-driven bean. You will need to add the references to the following entity beans:

- *applicant*—Coded name `java:comp/env/ejb/ApplicantLocal`
- *applicantSkill*—Coded name `java:comp/env/ejb/ApplicantSkillLocal`
- *job*—Coded name `java:comp/env/ejb/JobLocal`

- *jobSkill*—Coded name `java:comp/env/ejb/JobSkillLocal`
- *matched*—Coded name `java:comp/env/ejb/MatchedLocal`

Step 6—Testing the `ApplicantMatch` Bean

Run the Agency application using the appropriate `runAll` batch file for your operating system. Use the Register screen to add a new applicant, whose location is London and skills are Cigar Maker.

Use the `Tables screen` to view the contents of the `Matched` table and check that a row has been added for the new applicant with the following details:

- Job—Cigar trimmer
- Customer—winston
- Exact—false

Add another applicant whose location is also London but and whose skills are Cigar Maker and Critic. Check that this creates a row with the following details in the `Matched` table:

- Job—Cigar trimmer
- Customer—winston
- Exact—true

Change the skills for this second applicant. Remove the Cigar Maker and Critic and add the skill Bodyguard.

Check that the row for this applicant has now been deleted from the `Matched` table.

If these checks are okay, congratulations! You have successfully deployed the `ApplicantMatch` Message-driven bean. Of course, you can add or amend other applicants to find other job matches in the system.

Using Other Architectures

Message-driven beans were designed to operate within the context of JMS implementations with messages sent by a JMS Server. This does not mean Message-driven beans can not process messages sent by e-mail, HTTP, FTP, or any other protocol. As long as the server is able to convert these protocols into a JMS message (simple encapsulation will normally do), it can be handled by a Message-driven bean.

If the messages are defined in an open, extensible language like XML, unparalleled interoperability can be achieved in loosely-coupled systems using a model that is easy to

understand. This means that Message-driven beans have the potential to become the de-facto model for handling any message type.

Summary

Today, you have created some simple Message-driven beans and seen how easy it is consume JMS messages. Message-driven beans are a useful addition to the existing entity and session EJBs, offering a way for you to write asynchronous message consumers. You have seen how the container manages the life cycle of the bean, its transactions and security, so that having deployed your Message-driven beans, you can forget about them.

Q&A

Q What are the major differences between Message-driven beans and Entity or Session beans?

A Message-driven beans have no client interface; they have no `Home`, `Local`, or `Remote` interfaces—their methods cannot be called directly. Message-driven beans exist only to consume JMS messages and are controlled by the container. They are anonymous and are called asynchronously. They do not have a passive state. They have no client security context.

Q What are the two interfaces a Message-driven bean must implement?

A The `javax.ejb.MessageDrivenBean` interface and the `javax.jms.MessageListener` interface.

Q What is the Method Ready Pool?

A The Method Ready Pool is the collection of Message-driven bean instances that are available in the container to consume JMS messages.

Q How can I associate a Message-driven bean with a queue or a topic?

A A queue or topic is associated with a Message-driven bean at deploy time. The queue or topic must have already been registered with the J2EE system.

Exercise

To extend your knowledge of the Message-driven beans, try the following exercise.

1. Extend the Agency case study. Add a Message-driven bean that receives a message from the `AdvertiseJob` Session bean when a new job is advertised. The Message-driven bean should search through all the applicants to find those suitable to be considered for the job. To be considered for a job, the applicant must match the job's:

- Location
- At least one skill

If the applicant has all the required skills set `exactMatch` to `true`; otherwise, `false`.

All applicants that match at least one skill must be added to the `Matched` table.

Don't forget to create a JMS queue for the messages (you can't use the same queue as the one used in the Applicant example).

Add some new jobs and use the `TableClient` program to check that rows are being added to the `Matched` table (this will only happen if are some applicants that match the job's location and skills).

2. Extend your previous solution to update the `Matched` table when job adverts are changed. Hint: you can delete the old matched rows and add the applicants that match the new criteria rather than try to update the rows.

   For completeness you should update the `Register` and `Agency` beans so that when a job or customer is removed, all their entries in the `Matched` table are also removed. The `Matched` Entity bean has suitable home methods that support this functionality.

WEEK 2

DAY 11

JavaMail

In Day 9, "Java Message Service," and Day 10, "Message-Driven Beans," you learned about the Java Messaging Service and Message-driven beans. These technologies allow you to write code that provides application-to-application communication. Today's lesson looks at how you can provide application-to-human communication through the use of e-mail.

Today you learn how e-mail systems work and how the JavaMail API models these systems. You will then explore the API's main classes and see how to use these to provide typical day-to-day e-mail functionality, such as sending attachments and deleting messages. Finally, you will have the opportunity to expand on the case study you have been developing so that it sends e-mail messages to people who register with the service.

Today's lesson covers the following topics:

- Understanding e-mail systems, formats, and protocols
- Creating and sending plain text e-mail messages
- Creating and sending multi-media e-mail messages
- Sending e-mail messages with attachments

- Retrieving e-mail messages and attachments
- Deleting e-mail messages on the server
- Authenticating users and security

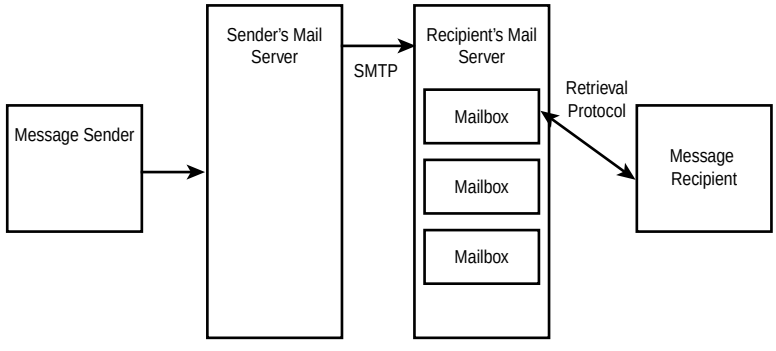
Understanding E-Mail

E-mail is something that most people take for granted without ever really understanding how it works. If you want to write applications that send and receive e-mail messages, it is essential to have some understanding of a typical e-mail system environment.

E-mail messages are sent on a client/server basis, but one that is different to that used for Web pages. Figure 11.1 shows a typical e-mail delivery process. As you can see, both the sender and recipient act as clients to e-mail servers. The sender creates a message and this forwards to an e-mail server. The server then uses the Simple Mail Transfer Protocol (SMTP) to send the message across the Internet to the recipient's mail box on another e-mail server. The receiver then uses a retrieval protocol, such as Post Office Protocol (POP3) or Internet Message Access Protocol (IMAP), to retrieve the message from their e-mail server.

FIGURE 11.1

A simple e-mail architecture.



The actual e-mail message itself consists of two sections—the header and the body. Mail headers are name-value pairs that define certain attributes of a particular message, such as who the sent the message and when the message was sent. The body is the actual e-mail message. Originally, the message body could only contain ASCII-based text. The standard 128-character ASCII set and the inability to embed multimedia objects or attach files meant that e-mail was restrictive. Over the years, there have been a number of ways to expand the functionality of e-mail. Today, you can use the Multipurpose Internet Mail Extensions (MIME) format to construct and send content-rich e-mail messages that are

not limited by the standard 128-character ASCII set. You will learn more about MIME in just a moment, but first, today's lesson will provide you with an overview of some the commonly used e-mail protocols.

SMTP

Simple Mail Transfer Protocol (SMTP) is the protocol that mail servers use to send messages to one another. SMTP communication occurs over TCP port 25 using a simple sequence of four-character client commands and three-digit server responses. It is unlikely that *you* will ever have to communicate using SMTP directly with a mail server, but it might interest you to see a typical conversation between a client and a mail server:

```
HELO madeupdomain.com
    250 Hello host127-0-0-1.anotherdomain.com [127.0.0.1]
MAIL FROM: me@anotherdomain.com
    250 < me@anotherdomain.com > is syntactically correct
RCPT TO: user@madeupdomain.com
    250 < user@madeupdomain.com > is syntactically correct
DATA
    354 Enter message, ending with "." on a line by itself
Hello World!
.
    250 OK id=1643UJ-00030Z-00
QUIT
    221 closing connection
```


You can see just how simple the protocol is. The client connects to the server and issues a `HELO` command and the server responds with a `250` (OK) response. The client then issues a `MAIL FROM:` command and a `RCPT TO:` command and, in both instances, the server replies with a `250` response. The client then issues a `DATA` command and sends a message. Finally, the server issues a `250` response and the client issues the `QUIT` command.

The important thing to note about SMTP is that it is not used to deliver a message directly to the recipient but, instead, delivers it the recipient's mail server. This mail server then forwards the message to the recipient's mail box—a file or other repository that is held on the server—and not the recipient's client machine.

Post Office Protocol 3 (POP3)

POP3 is a protocol that allows message recipients to retrieve e-mail messages stored in a mailbox on a mail server. The protocol operates on TCP port 110 and, like SMTP, uses a series of simple requests and responses. Unlike SMTP, users must provide authentication credentials (username and password) before they can download e-mail messages from their mail boxes.

Many e-mail clients use POP3, although the protocol allows quite limited server-side manipulation of messages. On the server, the user may list, delete, and retrieve e-mail messages from his or her mail box.

Internet Message Access Protocol (IMAP)

Like POP3, IMAP is an e-mail message retrieval protocol. Also like POP3, it works on a simple request-response model, but it does operate on a different TCP port—port 143. The biggest difference between the two protocols is that IMAP transfers a lot of client-side functionality to the server. For example, you can browse messages' subjects and sizes and senders' addresses before you decide to download the messages to your local machine. You can also create, delete, and manipulate folders on the server, and move messages between these folders.

Other Protocols

The three previously mentioned protocols are prevalent current e-mail systems, but there are other e-mail protocols. Some of these are previous versions of existing protocols. For example, some machines might still run POP2 servers. Other protocols are variations on those previously mentioned—for example, IMAP over SSL. Finally, there are a variety of protocols that provide either specialist functionality or different interpretations on the popular protocols. For example, the Network News Transport Protocol (NNTP) is the main protocol clients and servers use with Usenet newsgroups. If you want to learn more about this protocol, refer to Request for Comments (RFC) 997, which is available at `http://www.rfc.net/rfc997.html`.

Multipurpose Internet Mail Extensions (MIME)

The MIME format extends the capabilities of e-mail beyond the 128-character ASCII set to provide

- Message bodies in character sets other than US-ASCII
- Extensible formats for non-textual message bodies, such as images
- Multipart message bodies (you'll learn about these later today)
- Header information in character sets other than US-ASCII
- Unlimited message body length

The MIME standard provides a standard way of encoding many different types of data, such as GIF images and MPEG videos. MIME defines additional message headers that a client can then use to unpack, decode, and interpret the data the message body contains. The message body may consist of several body parts including attachments. The

"Creating Multi-Media E-mails" section of today's lesson explores MIME in more depth. You can also find out more about MIME by reading RFCs 2045 through to 2049, which you can access at `http://www.rfc.net/`.

Introducing the JavaMail API

As you have seen, e-mail systems have relatively complex architectures and use a selection of transport protocols. In the past, a developer wanting to send or retrieve e-mail messages would have to use TCP sockets and, using an appropriate protocol, talk directly to an e-mail server. As you can imagine, coding such applications was typically involved and intensive. Alternatively, a developer would have to use a vendor-specific API to access e-mail functionality, locking their code into one platform or technology. The JavaMail API changes all of this.

The API provides a generic model of an e-mail system, which allows you to send and retrieve messages in a platform independent and protocol independent way. In addition, JavaMail allows you to simply create different types of messages, such as plain text, those with attachments, or those with mixed binary content. Sun Microsystems' reference implementation of JavaMail supports the three most popular e-mail protocols—SMTP, POP3, and IMAP. Other protocols are available separately, for example, there are third-party implementations that support IMAP over SSL and NNTP.

Setting up Your Development Environment

If you downloaded and installed the J2EE reference implementation, you are ready to start exploring the JavaMail API. If you did not install the reference implementation, you will need to install the JavaMail API and Java Activation Framework (JAF) before you can start writing code.

Before you download the JavaMail API, ensure that you have J2SE 1.2.X or later correctly installed. You must also install the Java Activation Framework (JAF) because the JavaMail API requires it. The API requires the framework to handle arbitrary large blocks of data (you will learn more about this later in today's lesson). You can download JAF from `http://java.sun.com/products/javabeans/glasgow/jaf.html`.

Both JAF and the JavaMail API are written in pure Java, so they will operate on any platform running JDK 1.1 or later.

After you have downloaded the framework, installation is straightforward. Simply unzip the download file and append the location of `activation.jar` to your class path. To do this on a Windows platform, issue the following command:

```
set CLASSPATH=%CLASSPATH%;C:\jaf-VERSION\activation.jar
```

On a Unix-based system that has a Bourne or Korn shell, use the following:

```
$CLASSPATH=$CLASSPATH:usr/java/jaf-VERSION
$export $CLASSPATH
```

You have now installed JAF; now download the JavaMail API from `http://java.sun.com/products/javamail/index.html`. When downloaded, the actual installation of JavaMail is as simple as installing JAF. Simply unzip the download file and append the location of `mail.jar` to your class path.

That's it! You have successfully installed the JavaMail API. You will find a selection of demonstration applications in the demo directory under the JavaMail installation directory. In addition, you can find the API documentation in the `javadocs` directory, which is also under the installation directory.

Sending a First E-mail

This section introduces you to the core classes you need to send an e-mail message. Specifically, you will use these classes to write code that accepts a SMTP host and mail recipient from the command line, and then sends a plain text e-mail to that recipient.

Although the application uses the command line, you can modify the code for use in other situations if you want. For example, you could modify the code so that it accepts parameters from another application or presents the user with a desktop or applet GUI written using Swing or AWT.

Creating a First E-mail

Now that you have set up your development environment, you can write the code that will send an e-mail message. This application uses classes from the `javax.mail` package and the `javax.mail.internet` package. The `javax.mail` package provides classes that model a mail system, for example all mail systems require messages to contain a message body. The `javax.mail.internet` package provides classes that are specific to Internet mail systems, for example, a class that allows you to manipulate the content type of a message. This application commences by importing both of these packages and `java.util.Properties`, which allows you to set the SMTP host as a system property.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
```

Because this is a simple application, the `main()` method contains all the code to collect user input and send the e-mail message. In this example, the user will pass three parameters to the application via the command line—SMTP host, recipient's e-mail address, and sender's e-mail address:

```
public class SendMail {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: SendMail SMTPHost ToAddress FromAddress");
            System.exit(1);
        }

        String smtpHost=args[0];
        String to=args[1];
        String from = args[2];
```

Before you can create and send an e-mail message, you must have authenticated access to a SMTP host. If you are unsure of your SMTP host, either check with your system administrator or obtain the host address from the configuration of an e-mail application. To help you identify the host, the naming conventions for hosts means that your SMTP host is likely to take the form of `smtp.host` or `mail.host`. After you identify the SMTP host, you must make it available as a system property to the JVM. To do this, you must add the host name to the system properties list:

```
Properties props = System.getProperties();
props.put("mail.smtp.host",smtpHost);
```

The first line of code gets a `Properties` object that represents the system properties. The code then writes a key/value pair to the system properties list. The first argument of the `put()` method, `mail.smtp.host`, is defined by Appendix A of the JavaMail specification. Table 11.1 shows all of the properties the specification defines.

TABLE 11.1 Mail Environment Properties

Property	*Description*
`mail``.store.protocol`	Specifies the default message access protocol. The default value is the first appropriate protocol in the mail configuration files.
`mail.transport.protocol`	Specifies the default mail transport protocol. The default value is the first appropriate protocol in the mail configuration files.

TABLE 11.1 Continued

Property	*Description*
`mail.host`	Specifies the default mail server. The default value is the local machine.
`mail.user`	Specifies the default username used when connecting to the mail server. The default value is `user.name`.
`mail.protocol.host`	Specifies the protocol-specific mail server. This property overrides any value you specify for `mail.host`. The default value is `mail.host`.
`mail.protocol.user`	Specifies the protocol-specific username used when connecting to the mail server. This property overrides the value of `mail.user`. The default value is `mail.user`.
`mail.`*`from`*	Specifies the return address of the current user. The default value is `username@host`.
`mail.debug`	Specifies whether debug is enabled or disabled. The default value is false—debug is disabled.

In this example, you overrode the value of `mail.protcol.host`, which in turn overrides the value of `mail.host`.

Now that the code defines a `Properties` object containing the SMTP host, you create a mail session. A `Session` object represents the mail session, and all e-mail functionality works through this object. The `Session` object has two private constructors. The first provides a unique mail session that applications cannot share. To get this type of session you call the `getInstance()` method:

```
Session session = Session.getInstance(props, null);
```

In this example, the first argument is the `Properties` object that you previously created. The second argument, `null`, is an `Authenticator` object, which today's lesson discusses later in the "Authenticating Users and Security" section. The second constructor also takes these arguments, but it differs in that it creates a `Session` object that applications can share. It is this type of `Session` your first e-mail application uses:

```
Session session = Session.getDefaultInstance(props, null);
```

After you create a `Session`, you can start creating an e-mail message. The `Message` class represents an e-mail message, but it is an abstract class, so you must implement a subclass. The JavaMail API defines only one subclass, `MimeMessage`, that represents a standard MIME style e-mail message. The class has five constructors, but you will only use one of these at the moment—the default constructor. This accepts a single parameter—a `Session` object:

```
MimeMessage message = new MimeMessage(session);
```

This creates an empty `MimeMessage` object. You must now fill the object with appropriate attributes and content—in this example, subject, message text, from address, and to address. The `MimeMessage` class exposes a large number of methods for getting and settings these attributes. Today, you will explore several of these methods, but for a complete guide, refer to the API documentation. In this example, you first set the text of the message by using the `setText()` method:

```
message.setText("Hi from the J2EE in 21 Days authors");
```

The `setText()` method not only sets the message's text, but it also sets the content type of the message to `text/plain`. If the content is not of the type `text/plain`, you must first use the `setContent()` method to set the message's content type—you will learn how to do this later today in the "Authenticating Users and Security" section. The `setText()` method throws an exception of the type `MessagingException`, which is the base class for all messaging class exceptions. The majority of the methods associated with the mail packages throw this exception or one of its subclasses, so inevitably you have to catch these exceptions.

To set the subject of the message, you use the `setSubject()` method, which takes a string representing the subject as a parameter:

```
message.setSubject("Hi!");
```


The message now requires the addresses of the recipient and sender. To set the sender's address, you use the `setFrom()` method. The method takes a single argument of the type `Address`. The `Address` class represents an address in an e-mail message. The class is abstract, so you must use one of its subclasses, namely `InternetAddress`. This class has four constructors; the one you use takes a single parameter—an e-mail address as a string:

```
message.setFrom(new InternetAddress(from));
```

You provide the recipient's address to the `MimeMessage` object in a similar way. But you must also stipulate to which type of recipient the address relates. The `Message` class has one inner class, `RecipientType`, that defines the type of recipient within an e-mail message. The class exposes three fields that relate to the different recipient types:

- *BCC*—Blind carbon copy recipients
- *CC*—Carbon copy recipients
- *TO*—TO recipients

In this application, you only send the message to one recipient, so you use the `TO` field. When you set the from address, you used the `setFrom()` method, and you use the `addRecipient()` method to set the To address. The code to set the recipient's address appears as follows:

```
message.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
```

You have now created the message, now you must send the message. The `Transport` class, which models an e-mail transport system, provides you with the means to send the message. The class provides two static `send()` methods that send messages. The first form takes a `Message` object and an array of `Address` objects as arguments. It sends the message to all the recipients the `Address` objects define and ignores any addresses the message defines. The second version of this method takes one parameter—a `Message` object. Note that the `Message` object must define at least one recipient's address. It is this method you will now use:

```
Transport.send(message);
```

The `send()` method may throw a `MessagingException` or a `SendFailedException`. The `SendFailedException` is thrown if the `send()` method fails to send one or more messages. The exception traceback includes a list of all the e-mail addresses to which it could not send messages.

You have now completed the code to send an e-mail. Listing 11.1 shows the complete code:

LISTING 11.1 `SendMail.java` Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class SendMail {
    public static void main(String[] args) {
        if (args.length!=3) {
            System.out.println("Usage: SendMail SMTPHost ToAddress
➥FromAddress");
            System.exit(1);
        }

        String smtpHost = args[0];
        String to = args[1];
        String from = args[2];

        // Get properties object
        Properties props = System.getProperties();

        // Define SMTP host property
        props.put("mail.smtp.host",smtpHost);

        try {
```

LISTING 11.1 Continued

```
            // Get a session
            Session session = Session.getDefaultInstance(props,null);

            // Create a new message object
            MimeMessage message = new MimeMessage(session);

            // Populate message object
            message.setText("Hi from the J2EE in 21 Days authors.");
            message.setSubject("Hi!");
            message.setFrom(new InternetAddress(from));
message.addRecipient
➥(Message.RecipientType.TO,new InternetAddress(to));
            System.out.println("Message Sent");

            // Send the message
            Transport.send(message);
        }
        catch (MessagingException me) {
            System.err.println(me.getMessage());
        }
    }
}
```


To run this application, you must first compile it. Compile it from the command line using `javac` as you would for any other Java application. After it compiles, run the application by issuing the following command:

```
java SendMail mail.yourSMTPHost.com toAddress fromAddress
```

When the program executes, a `Message Sent` message should display. If the program fails to execute correctly, check the error messages the JVM displays. The most likely cause of failure is an incorrect SMTP host or one that is inaccessible from your system. There are several ways you can check whether the host exists and is visible. For example, you can run a trace route to your SMTP host; under Windows (all versions) issue the following command at the command prompt:

```
tracert yourSMTPHost
```

On Unix or Linux systems, use the following:

```
traceroute yourSMTPHost
```

If the host does not exist or is inaccessible, you will receive a message indicating that your trace route program could not resolve the hostname you specified, or on some systems, the displayed route will suddenly die.

Creating Multi-Media E-mails

Most modern e-mail clients have the capability of displaying formatted HTML messages. Many organizations now prefer to send HTML messages because they provide a richer viewing experience for the recipient. For example, a HTML message might incorporate an organization's logo and format the message's text so that it adheres to the organization's style preferences (font face, heading colors).

In the following example, HTML is used as the multi-media content. However, there is no intention to teach you HMTL (if you do not already know it), so the HTML is supplied for you.

The application you will now write is a typical example of an application an organization might use to send a marketing or informational e-mail to customers or potential customers. The application sends a HTML message to the recipient that contains an image file representing the organization's logo. To view the formatted output of the message, you must have an e-mail client that is capable of displaying HTML, such as Netscape Communicator, Microsoft Outlook, or a Java application that uses Swing `JEditorPane`.

There are two main ways of creating an HTML message that will display an image when viewed in an e-mail client. The simplest is when the HTML code references a URL that points to a publicly visible image location. This approach is simple to code, but does require that the image remains publicly visible for the potential lifetime of the message. In addition, it requires the e-mail client either to always have an external connection or cache a local copy of the images. The second approach is when you integrate the images into the message. This approach ensures that the recipient always sees the images. However, it does entail sending larger messages, which places higher loads on your server and the client's mail server. Today's lesson shows you both of these approaches starting with the external reference approach.

Creating the Message: Approach #1

The code for this approach to sending a HTML message is very similar to the code that sends a plain text message. The start of the code has a few additions. First, the code imports the `java.io` package to provide the classes that read the HTML file. Second, the code declares two extra variables—one defines the content type of the message and the other names the HTML source file.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

```
public class SendHTMLMail {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: SendHTMLMail SMTPHost ToAddress FromAddress");
            System.exit(1);
        }
        String smtpHost = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";
        String htmlFile = "HTMLSource1.html";
```

In common with the plain text message, you create `Properties`, `Session` and `Message` objects. You also set the subject and sender's and recipient's addresses.

```
    Properties props = System.getProperties();
    props.put("mail.smtp.host",smtpHost);
    Session session = Session.getDefaultInstance(props,null);
    MimeMessage message = new MimeMessage(session);
    message.setSubject("Hi!");
    message.setFrom(new InternetAddress(from));
    message.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
```

Adding the HTML to the `MimeMessage` object is the first significant change to the code. Because the HTML is stored in a file, you must first read the file's contents. The following code does this; as you can see, it simply uses the `java.io` classes:

```
    String msg="";
    String line=null;
    FileReader fr = new FileReader(htmlFile);
    BufferedReader br = new BufferedReader(fr);
    while ((line=br.readLine())!=null) {
        msg+=line;
    }
```

The code that sent a plain text message used the `setText()` method to set the message text. In this example, you use another method—`setContent()`. This method takes two parameters—an object that represents the message body and a string that defines the MIME type of the message body. This method allows you to stipulate the MIME type of the message, whereas the `setText()` method always applies a MIME type of `text/plain`. In the application you are currently building, the `setContent()` method takes a first argument of a string (the HTML) and a second argument of a variable with the value `text/html`:

```
message.setContent(msg,contentType);
```

After you define the message body, you send the message using the `send()` method of the `Transport` class. This is exactly the same approach you took when sending a plain

text message, so that code fragment isn't shown here. Instead, Listing 11.2 shows the complete code for sending a HTML e-mail message.

LISTING 11.2 `SendHTMLMail.java` Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class SendHTMLMail {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: SendHTMLMail SMTPHost ToAddress FromAddress");
            System.exit(1);
        }

        String smtpHost = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";
        String htmlFile = "HTMLSource1.html";

        // Get properties object
        Properties props = System.getProperties();

        // Define SMTP host property
        props.put("mail.smtp.host",smtpHost);

        try {
            // Get a session
            Session session = Session.getDefaultInstance(props,null);

            // Create a new message object
            MimeMessage message = new MimeMessage(session);

            // Populate message object
            message.setSubject("Hi!");
            message.setFrom(new InternetAddress(from));
message.addRecipient
➥(Message.RecipientType.TO,new InternetAddress(to));

            // read the HTML source
            String msg="";
            String line=null;
            FileReader fr = new FileReader(htmlFile);
            BufferedReader br = new BufferedReader(fr);
            while ((line=br.readLine())!=null) {
```

LISTING 11.2 Continued

```
                msg+=line;
            }

            // Add the content to the message
            message.setContent(msg,contentType);

            // Send the message
            Transport.send(message);
        }
        catch (MessagingException me) {
            System.err.println(me.getMessage());
        }
        catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
    }
}
```

The Java code for this application is now complete, but you still need the HTML source file. It is not one of the aims of this book to teach you HTML. However, there are two important points to note about the HTML source for this example. The first is that the HTML is regular HTML; you change nothing to include it within an e-mail message. The second point concerns the image that the message HTML message displays. The HTML defines an absolute URL to the image file:


```
http://www.samspublishing.com/images/topnav/logo-bottom.gif
```

You must use an absolute URL because the message does not include the image itself. Retaining the image source at a remote location offers some advantages over including the image within the message:

- *Loads*—If messages do not include image data, they remain small. This places a lower load on the mail server, especially in the case of bulk mailings.
- *Maintenance*—The image source file might be modified or updated and, as long as the image's URI remains the same, that modification or update will affect the already-delivered message.

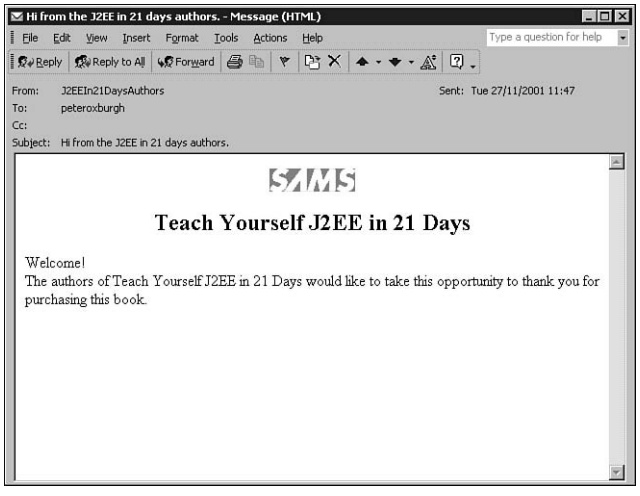
In contrast, this approach also has disadvantages, because the external image must always be accessible to the client or the client must cache a copy of the image source. If the client is unable to do either of these things, the image will not render within the message.

To run this application, you must first compile it. When compiled, run the application by issuing the following command:

```
java SendHTMLMail mail.yourSMTPHost.com toAddress fromAddress
```

Remember that you must have an e-mail client that is capable of viewing HTML messages to view the output from this application. Figure 11.2 shows the message sent by this application:

FIGURE 11.2
Sample output for Listing 11.2.


Hi from the J2EE in 21 days authors. - Message (HTML)
File Edit View Insert Format Tools Actions Help
Type a question for help
Reply Reply to All Forward
From: J2EEIn21DaysAuthors
Sent: Tue 27/11/2001 11:47
To: peteroxburgh
Cc:
Subject: Hi from the J2EE in 21 days authors.
SAMS
Teach Yourself J2EE in 21 Days
Welcome!
The authors of Teach Yourself J2EE in 21 Days would like to take this opportunity to thank you for purchasing this book.

Creating the Message: Approach #2

The second approach to sending a HTML e-mail message with an embedded image includes the image within the message itself. To create this type of message, you create a message that consists of multiple parts, where the HTML forms one part and the image a second part. This type of message is known as a multi-part message, and the next section of today's lesson explores these in some detail.

Writing the Code

Like the previous application, the code for this application creates `Properties`, `Session`, and `Message` objects and sets the message subject, recipient's address, and sender's address. The code only differs in how it creates the message body. For the sake of brevity, this part of today's lesson only walks through the code that differs from the previous application, but you can still see the full code in Listing 11.3.

As a result of creating a multi-part message, there are two small additions to the start of the code. You must import the `javax.activation` package (you'll learn why a little later) and create one additional variable with a value of the image file location:

```
String imageFile = "sams.gif";
```

In the previous application you set the message's body text by using the `setContent()` method, where you supplied a string containing HTML and a MIME type. The following is the line of code that did this:

```
message.setContent(msg,contentType);
```

To create a multi-part message, you must replace this line of code. The new code must define each of the parts of the message and then associate them with the message. Figure 11.3 shows the process of creating multi-part content.

FIGURE 11.3

Creating a multi-part message.

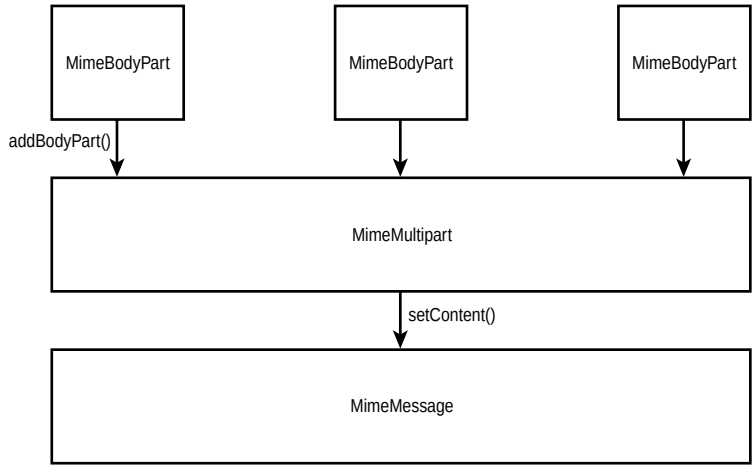


Figure 11.3 shows that you create `BodyPart` objects, each of which contains one part of the message's body. For example, in this application one part contains HTML and another part contains the image data. As you create each `BodyPart`, you add it to a `MimeMultipart` object by using its `addBodyPart()` method. After you have added all the `BodyPart` objects to the `MimeMultipart` object, you add this object to the `MimeMessage` object using its `setContent()` method. To write this as code, you start by creating a new `BodyPart` object:

```
BodyPart messageBodyPart = new MimeBodyPart();
```

In this example, the code creates an empty `BodyPart` object. However, the class offers two other constructors. The first creates accepts an input stream as an argument. The input stream is read and parsed, and then becomes the content of the `BodyPart` object. The final constructor accepts two arguments—an `InternetHeaders` object and a byte array of content. The `InternetHeaders` object contains the Internet headers that relate to

the given content. In most instances, you will not directly code the Internet headers; service providers primarily use the object.

You must now set the content of this first `BodyPart`. In this example, the content is the HTML that you read from a file and assigned to the variable `msg`. The `setContent()` method sets the content of a body part:

```
messageBodyPart.setContent(msg, contentType);
```

You may notice that this `setContent()` method takes the same arguments as the `setContent()` method you used with the `MimeMessage` object. Both the `MimeMessage` class and `BodyPart` class subclass classes, which themselves implement the `Part` interface. This interface is common to all messages and `BodyParts`. The purpose of the interface is to define attributes, common to all mail systems, and then provide accessor methods for these attributes. For example, these attributes include headers and content. For a complete reference to this interface, refer to the API documentation, which is available locally in the docs directory under the J2EE installation directory or online at `http://java.sun.com/products/javamail/1.2/docs/javadocs/overview-summary.html`.

Now that you have created the first body part, you must create a `MimeMultipart` object and add the body part to it. `MimeMultipart` has three constructors—the default constructor creates an empty `MimeMultipart` object. This object has a content type of `multipart/mixed` and is given a unique boundary string that you can access through the `contentType` field. The second constructor accepts a `DataSource` object as an argument, you'll learn more about `DataSource` objects a little later in this section. The final constructor accepts a single string that represents a given media subtype. There are a wide range of possible subtypes, such as `digest`, `encrypted`, and `related`. For further details about subtypes, refer to RFC 822, which is available at `http://rfc.net/rfc822.html`. The `related` subtype applies to objects that consist of multiple interrelated parts. It allows you to link these parts to present a single compound object. It is this subtype that you create with the following:

```
MimeMultipart multipart = new MimeMultipart("related");
```

Now you must add the body part, which contains the HTML, to the `MimeMultipart` object you just created. The `MimeMultipart's addBodyPart()` method provides two ways for adding the body part. The first way takes two arguments—a `BodyPart` and an index (`int`) to position the object. The seconds version of the `addBodyPart()` methods simply takes a `BodyPart` as an argument and then appends this to list of body parts it currently contains:

```
multipart.addBodyPart(messageBodyPart);
```

Now you must create a new `BodyPart` object that contains the image and then append this to the `MimeMultipart` object. In common with adding the HTML, you first create a new `BodyPart`:

```
messageBodyPart = new MimeBodyPart();
```

To add the image to the `BodyPart` object, you use the `setDataHandler()` method. This method accepts one argument—a `DataHandler` object. This object provides a consistent interface to several different data sources and formats, for example, an image or an input stream. In this example, the `DataHandler` object will reference a `DataSource` object that encapsulates the image file:

```
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
```

The final step before adding this body part to the `MimeMultipart` object is to set a header for the part. You need to set the header because your HTML source will reference the header to include the image within the message rather than reference the image via an absolute URL. You use the body part's `setHeader()` method to set the header:

```
messageBodyPart.setHeader("Content-ID","image1");
```


You can see that the method accepts two parameters. The first parameter is a string that represents an RFC 822 header value. The second parameter is the header's value that you will later reference from the HTML source.

That's it. You have created the body part that contains the image the message will display. Now you must add the body part to the `MimeMultipart` object that you created earlier. You add the body part in exactly the same way you added the first body part (the one containing the HTML source):

```
multipart.addBodyPart(messageBodyPart);
```

If you refer back to Figure 11.3, you can see that you must set the content of the message with the `MimeMultipart` object. As with the previous examples, you use the `setContent()` method to do this:

```
message.setContent(multipart);
```

The code that sends the message is identical to all the previous examples. Listing 11.3 shows the complete code for this application.

LISTING 11.3 SendMultiPartMail.java Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;

public class SendMultiPartMail {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: SendMultiPartMail SMTPHost ToAddress FromAddress");
            System.exit(1);
        }

        String smtpHost = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";
        String htmlFile = "HTMLSource2.html";
        String imageFile = "sams.gif";

        // Get properties object
        Properties props = System.getProperties();

        // Define SMTP host property
        props.put("mail.smtp.host",smtpHost);

        try {
            // Get a session
            Session session = Session.getDefaultInstance(props,null);

            // Create a new message object
            MimeMessage message = new MimeMessage(session);

            // Populate message object
            message.setSubject("Hi!");
            message.setFrom(new InternetAddress(from));
message.addRecipient
➥(Message.RecipientType.TO,new InternetAddress(to));

            // read the HTML source
            String msg="";
            String line=null;
            FileReader fr = new FileReader(htmlFile);
            BufferedReader br = new BufferedReader(fr);
            while ((line=br.readLine())!=null) {
                msg+=line;
            }
```

LISTING 11.3 Continued

```
            // Create your new message part
            BodyPart messageBodyPart = new MimeBodyPart();
            messageBodyPart.setContent(msg, contentType);

            // Create a related multi-part to combine the parts
            MimeMultipart multipart = new MimeMultipart("related");
            multipart.addBodyPart(messageBodyPart);

            // Create part for the image
            messageBodyPart = new MimeBodyPart();

            // Fetch the image and associate to part
            DataSource fds = new FileDataSource(imageFile);
            messageBodyPart.setDataHandler(new DataHandler(fds));
            messageBodyPart.setHeader("Content-ID","image1");

            // Add part to multi-part
            multipart.addBodyPart(messageBodyPart);

            // Associate multi-part with message
            message.setContent(multipart);

            // Send the message
            Transport.send(message);
        }
        catch (MessagingException me) {
            System.err.println(me.getMessage());
        }
        catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
    }
}
```


The Java code for this application is now complete, but you still need the HTML code. The HTML is identical to that used in the previous example with one exception. In the previous example, you referenced the image with an absolute URL. In this application, you use a CID (Content ID) URL and the value of the header you set for the body part that contains the image, for example,

```
<img src="cid:image1">
```

To run this application, you must first compile it. When compiled, run the application by issuing the following command:

```
java SendMultiPartMail mail.yourSMTPHost.com toAddress fromAddress
```

The message sent by this application should appear identical to that sent in the previous application, as shown in Figure 11.2.

Sending E-mails with Attachments

In the previous example, you used MIME multi-part messages to send a HTML message that displays an image. Another use of MIME is to send message attachments, such as MPEG videos, JPEG images, and plain text files. In this example, you will create a small command line application that sends a message with an attached XML file to a named recipient.

The code for this application is very similar to the previous applications. Today's lesson only explores those sections of code that differ from the previous examples, but you can still find a full listing at the end of this section or on the CD-ROM that accompanies this book.

This application starts by creating `Properties`, `Session`, and `MimeMessage` objects. It also adds the message subject, sender's address, and recipient's address to the `MimeMessage` object. You must then create an empty `BodyPart` object, using the same technique as the previous example:

```
BodyPart messageBodyPart = new MimeBodyPart();
```

This is the first part of the message. You will create another part in a moment that will contain the file attachment. Before you do this, use the `setText()` method you used in previous examples to set the message's body text:

```
messageBodyPart.setText("Here's an attachment!");
```

Now you must create a new `MimeMultipart` object to hold the two parts of the message. In the previous example, you passed an argument of `related` (the MIME subtype) to the `MimeMultipart` constructor. This was because the different body parts connected to form a compound object, but in this example, the attachment does not integrate with the message body. The subtype in this example is `multipart/mixed`. This subtype is the default for the constructor, so you do not have to explicitly pass it to the constructor:

```
Multipart multipart = new MimeMultipart();
```

Now that you have created the `MimeMultipart` object, you can add the first of the body parts to it. Simply use the `addBodyPart()` method, as you did in the previous application:

```
multipart.addBodyPart(messageBodyPart);
```

That is the first part of the message. You must now create the body part that contains the attachment and add it to the `MimeMultipart` object. The previous example set a data han-

dler for the body part; the data handler referenced a `DataSource` object that encapsulated an image file. You use exactly the same approach with an attachment:

```
// Create an empty body part
messageBodyPart = new MimeBodyPart();

// Create a new DataSource, passing it the attachment file
DataSource source = new FileDataSource(fileName);

// Set the data handler
messageBodyPart.setDataHandler(new DataHandler(source));
```

The final step before adding the body part to the `MimeMultipart` object is to set the filename that associates with this body part. To do this, you use the `MimeBodyPart`'s `setFileName()` method that takes one argument, a string representing the filename. One other point to note about this method is that it might throw one of three exceptions: `MessagingException`, `javax.mail.IllegalWriteException` (descended from `MessagingException`), or `java.lang.IllegalStateException`. An `IllegalWriteException` occurs when the underlying implementation can be modified, and an `IllegalStateException` occurs when the body part is obtained from a read only directory.

```
// associate the file name
messageBodyPart.setFileName(fileName);
// add body part to MimeMultipart object
multipart.addBodyPart(messageBodyPart);
```

The remaining code is identical to the previous example. Listing 11.4 shows the complete code for this application.

LISTING 11.4 `SendAttachmentMail.java` Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendAttachmentMail {
    public static void main(String[] args) {
        if (args.length!=4) {
System.out.println
➥("Usage: SendAttachmentMail SMTPHost
➥ToAddress FromAddress AttachmentName");
            System.exit(1);
        }

        String smtpHost = args[0];
        String to = args[1];
```

LISTING 11.4 Continued

```
            String from = args[2];
            String contentType = "text/html";
            String fileName = args[3];

            // Get properties object
            Properties props = System.getProperties();

            // Define SMTP host property
            props.put("mail.smtp.host",smtpHost);

            try {
                // Get a session
                Session session = Session.getDefaultInstance(props,null);

                // Create a new message object
                MimeMessage message = new MimeMessage(session);

                // Populate message object
                message.setSubject("Hi!");
                message.setFrom(new InternetAddress(from));
    message.addRecipient
    ➥(Message.RecipientType.TO,new InternetAddress(to));

                // Create the message body part
                BodyPart messageBodyPart = new MimeBodyPart();
                messageBodyPart.setText("Here's an attachment!");
                Multipart multipart = new MimeMultipart();
                multipart.addBodyPart(messageBodyPart);

                // Create the attachment body part
                messageBodyPart = new MimeBodyPart();
                DataSource source = new FileDataSource(fileName);
                messageBodyPart.setDataHandler(new DataHandler(source));
                messageBodyPart.setFileName(fileName);
                multipart.addBodyPart(messageBodyPart);

                // Put parts in message
                message.setContent(multipart);

               // Send the message
               Transport.send(message);
            }
            catch (MessagingException me) {
                System.err.println(me.getMessage());
            }
            catch (IllegalStateException ise) {
                System.err.println(ise.getMessage());
            }
        }
    }
```

The code for this application is now complete. To run this application, you must first compile it. When compiled, run the application by issuing the following command:

```
java SendAttachmentMail mail.yourSMTPHost.com toAddress fromAddress attachment
```

If you don't have a file to attach at hand, the CD-ROM accompanying this book contains a small XML file that is appropriate for this purpose.

Exploring the JavaMail API

So far, today's lesson has concentrated on sending e-mail messages, but the JavaMail API provides support for other common e-mail operations, such as retrieving messages and attachments and deleting messages on the server. The API provides numerous classes that allow you to perform these types of operations. The remainder of today's lesson gives you an introduction to these operations. For a complete reference to the API's interfaces, classes, and methods, refer to the API documentation.

Retrieving Messages

Earlier in today's lesson, you learned that there are a number of message retrieval protocols, and that most prevalent of these are POP3 and IMAP. The application you are about to build uses POP3 to retrieve messages from a mail server. Note that because you work with an abstraction of a mail system, the technique for accessing a mail box with IMAP is very similar.

Specifically, this application runs from the command line and accepts three parameters—a host, username, and password. You can see the code that accepts these parameters in Listing 11.5 at the end of this section. As with all the applications in today's lesson, you first create an empty `Properties` object and a `Session` object:

```
Properties props = new Properties();
Session session = Session.getDefaultInstance(props, null);
```

Figure 11.4 shows the remainder of the process for retrieving messages.

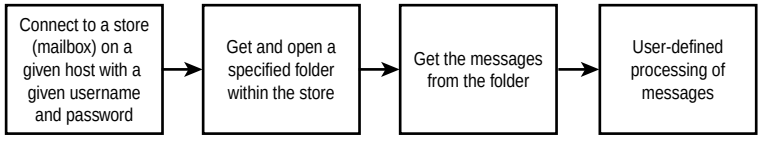

Connect to a store (mailbox) on a given host with a given username and password
Get and open a specified folder within the store
Get the messages from the folder
User-defined processing of messages

FIGURE 11.4
The message retrieval process.

The first step in the process is to connect to a store on a given host. To do this, you must first create a `Store` object by calling the `Session` object's `getStore()` method. This method accepts one parameter that specifies the retrieval protocol and returns a `Store` that implements this protocol. After you create a `Store`, you invoke its `connect()` method to connect to the remote message store. The `connect()` method takes three strings as arguments—`host`, `username`, and `password`.

```
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

When you use the JavaMail API to send POP3 credential information, the information is not encrypted. In fact, the API does not come with built-in support for credential encryption or the facility to use secure mail transport protocols, because it only supports IMAP, SMTP, and POP3. However, some third-party protocol providers do provide support for secure mail protocols, such as IMAP over SSL/TLS. You can find a list of these providers at `http://java.sun.com/products/javamail`.

After you connect to the message store, you open a message folder. To do this, create a `Folder` object. The `Folder` class, like the `Store` class, is abstract. To create an instance of the class, you invoke the `getFolder()` method of the `Store` object. The method takes one parameter, the name of the folder as a string. In this example, the folder name is `INBOX`:

```
Folder folder = store.getFolder("INBOX");
```

Now you can open the folder using the `open()` method of the `Folder` class. The method takes one argument that indicates the mode with which to open the folder. Static fields of the `Folder` class give the two possible values for the mode, `READ_ONLY` and `READ_WRITE`:

```
folder.open(Folder.READ_ONLY);
```

To retrieve the messages from the folder, you can either use the `Folder`'s `getMessage()`method or its `getMessages()` method. The `getMessages()` method is overloaded and, as such, there are three versions of this method, each of which returns an array of `Message` objects. The first version takes an array of `int`s and returns the messages at the indices the `int`s specifies. The second version takes a start index and an end index and returns the messages within this range. The final version takes no arguments and returns all the messages within a folder:

```
Message messages[] = folder.getMessages();
```

The final step in the message retrieval process is to perform some form of user-defined processing on the retrieved messages. In this example, the code simply iterates through the messages printing the subject, sender, time, and then printing the entire message.

```
for (int i=0; i<messages.length; i++) {
System.out.println(i + ": "
➥+ messages[i].getFrom()[0] + "\t"
➥+ messages[i].getSubject() + "\t"
➥+ messages[i].getSentDate() + "\n\n");
    messages[i].writeTo(System.out);
}
```

As you can see, the `Message` class exposes a number of methods that take the form `getITEM()`. The API documentation provides a complete list of these methods. The `Message` class implements the `Part` interface, and it is from this that it inherits the `writeTo()` method. This method outputs a byte stream to a specified output stream, which in this example is `System.out`. It is important to note that the `writeTo()` method can throw a `java.io.IOException` that you must catch.

That is the message retrieval process completed, but to complete the code you must close the resources it uses. In this example, you must close the `Folder` and `Store` objects. To do this, simply call their `close()` methods. You must pass the `Folder's close()` method a single Boolean parameter indicating whether to delete (expunge) any messages marked for deletion within the folder on the server. You will learn how to mark messages for deletion in the "Deleting Messages" section of today's lesson.

```
folder.close(false);
store.close();
```

Listing 11.5 shows the complete code for this application.

LISTING 11.5 `RetrieveMail.java` Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class RetrieveMail {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: RetrieveMail POPHost username password");
            System.exit(1);
        }
```

LISTING 11.5 Continued

```
            String host = args[0];
            String username = args[1];
            String password = args[2];

            try {
                // Create empty properties object
                Properties props = new Properties();

                // Get a session
                Session session = Session.getDefaultInstance(props, null);

                // Get the store and connect to it
                Store store = session.getStore("pop3");
                store.connect(host, username, password);

                // Get folder and open it
                Folder folder = store.getFolder("INBOX");
                folder.open(Folder.READ_ONLY);

                // Get messages
                Message messages[] = folder.getMessages();

                for (int i=0; i<messages.length; i++) {
    System.out.println(i + ": " + messages[i].getFrom()[0]
    ➥+ "\t" + messages[i].getSubject() + "\t"
    ➥+ messages[i].getSentDate() + "\n\n");
                    messages[i].writeTo(System.out);
                }

                // Close resources
                folder.close(false);
                store.close();
            }
            catch (MessagingException me) {
                System.err.println(me.getMessage());
            }
            catch (IOException ioe) {
                System.err.println(ioe.getMessage());
            }
        }
    }
```

To run the application, compile it and then issue the following command:

```
java RetrieveMail mailHost username password
```

Deleting Messages

In the previous example, you retrieved messages from a mail server without removing them after retrieval. Of course in a real-world situation, you could not leave messages on the mail server indefinitely; it would soon become very crowded! Instead, you would delete messages after retrieving them. To modify the previous example so that messages are deleted after retrieval is a straightforward process.

Previously, you opened the folder in `READ_ONLY` mode, but to retrieve and delete messages you must open the folder in `READ_WRITE` mode. So simply replace the line of code that opened the folder with the following line:

```
folder.open(Folder.READ_WRITE);
```

Deleting messages involves the use of flags. Messages can support a number of flags that indicate the state of a message. The `Flags.Flag` class supplies predefined flags that are accessible as static fields. Table 11.2 shows these flags and describes their meaning. Note, however, that just because these flags exist doesn't mean that all mail servers support them. In fact, POP3 mail servers typically only support the `DELETED` flag; IMAP servers normally support more of the flags. To find out exactly which flags your mail server supports, you can call the `getPermanentFlags()` method of a `Folder` object, which returns a `Flags` object that contains all the supported flags.

TABLE 11.2 Predefined Message Flags

Flag	*Description*
`ANSWERED`	The client has replied to the message.
`DELETED`	The message is marked for deletion.
`DRAFT`	The message is a draft.
`FLAGGED`	The client has flagged the message.
`RECENT`	The message has arrived in the folder since it was last opened.
`SEEN`	The message has been retrieved by the client.
`USER`	Indicates that the folder supports user defined flags.

To mark a message for deletion, you use the `setFlag()` method of the `Message` object. The method takes two parameters—a flag and a Boolean indicating the flag's value. For example, to mark a message ready for deletion in the previous example, you add the following line to the end of the `for` loop:

```
messages[i].setFlag(Flags.Flag.DELETED, true);
```

In case you are unsure, the modified `for` loop would appear as follows:

```
for (int i=0; i<messages.length; i++) {
System.out.println(i + ": "
➥+ messages[i].getFrom()[0] + "\t" + messages[i].getSubject()
➥+ "\t" + messages[i].getSentDate() + "\n\n");
    messages[i].writeTo(System.out);

    // Mark the message for deletion
    messages[i].setFlag(Flags.Flag.DELETED, true);
}
```

To complete the deletion, you must pass the folder's `close()` method a Boolean of `true`. Passing the value of `true` ensures that any messages marked for deletion are deleted (expunged) when the folder closes:

```
folder.close(true);
```

That's it. If you want to view the complete modified listing, you will find it on the CD-ROM accompanying this book.

Getting Attachments

Retrieving an attachment from a message is a more involved process than simply reading a normal message. If you remember from earlier, an attachment is a part of a multi-part message. When you retrieve a message that has an attachment, you must iterate through the body parts and identify which ones are attachments. After you identify a part as an attachment, you must write that part's content to a file.

You cannot simply identify a part as an attachment through its content type, because the sender of the message part may have intended for it to be displayed inline—like the HTML message you created earlier. Fortunately, RFC 2183 defines the `Content-Disposition` MIME message header. This header allows a message sender to mark body parts as either inline (displays within the message text) or attached (the part is an attachment). The JavaMail API provides support for this header, as you will soon learn.

The code for this application is based on the `RetrieveMail` application you wrote earlier. There are two changes to the code. The first is that it will only retrieve the first message on the mail server rather than all the messages. The second is that where it iterated through the messages and performed some basic processing on them (printed parts of the message), it will now process the individual parts of the message. The revised section appears as follows:

```
Message message = folder.getMessage(1);
    Multipart multipart = (Multipart)message.getContent();
    // Process each part of the message
    for (int i=0; i<multipart.getCount(); i++) {
```

```
            processPart(multipart.getBodyPart(i));
        }
```

In this application, you iterate through the messages and call a `processPart()` method that you will write shortly. The code passes one parameter to the method, a `Part` object. The `processPart()` method checks a body part to determine whether it is an attachment. In addition, it checks a body part that is identified as an attachment to determine if it has a filename. If it doesn't, the method provides a temporary file:

```
private static void processPart(Part part)
➥throws MessagingException, IOException{
    String disposition = part.getDisposition();
    String fileName = part.getFileName();
    if (disposition.equals(Part.ATTACHMENT)) {
        if (fileName == null) {
            fileName = File.createTempFile("attachment", ".txt").getName();
        }
        writeFile(fileName,part.getInputStream());
    }
    else {
        // It's not an attachment – provide appropriate processing
    }
}
```

The code starts by using the `getDisposition()` and `getFileName()` methods, which the `Part` interface defines, to get the part's disposition and filename. If either of these items does not exist, the methods return `null`. The code then checks whether the disposition is of the type `ATTACHMENT`; the other possible value is `INLINE`. If the disposition is of the type `ATTACHMENT`, the code checks to see if the body part has an associated filename. If it does not, the code assigns a temporary filename. In both instances, the code calls the `writeFile()` method that you will write shortly. If the body part does not have a disposition type of `ATTACHMENT`, the code assumes that the body part should be displayed inline. The premise here is that the disposition will be either `INLINE` (should display within the message) or `null`. If the disposition is `null`, taking into consideration that this is a multipart message, the message sender most likely wanted the part to be displayed inline.

The `writeFile()` method writes the content of a body part to a file. The method accepts two parameters—a string filename and an input stream:

```
private static void writeFile
➥(String fileName, InputStream in) throws IOException {
```

The method begins by checking that the named file does not already exist; you don't want to overwrite a file! The code that does this simply creates a `File` object and checks whether it exists. If the file exists, the code alters the filename (through a numeric increment) and then repeats the previous process:

```
    File file = new File(fileName);
    for (int i=0; file.exists(); i++) {
        file = new File(fileName+i);
    }
```

After the code identifies a valid filename, it writes the body part's content to the file. You may think that you could use the part's `writeTo()` method that you used in a previous example. You cannot use this method because it doesn't decode the attachment. Instead, the code uses familiar `java.io` classes that allow you to copy the body part's input stream onto a file output stream. This approach automatically decodes a variety of encoding formats, including Base-64:

```
    BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(file));
    BufferedInputStream bis = new BufferedInputStream(in);
    int aByte;
    while ((aByte = bis.read()) != -1) {
        bos.write(aByte);
    }
    bos.flush();
    bos.close();
    bis.close();
}
```

That's it. You have written the code to identify an attachment and write it to a file. Listing 11.6 shows the complete code listing for the `RetrieveAttachment` application. To run the application, compile the code and then run it by issuing the following command:

```
java RetrieveAttachment host username password
```

LISTING 11.6 `RetrieveAttachment.java` Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class RetrieveAttachment {
    public static void main(String[] args) {
        if (args.length!=3) {
System.out.println
➥("Usage: RetrieveAttachment host username password");
            System.exit(1);
        }

        String host = args[0];
        String username = args[1];
        String password = args[2];
```

LISTING 11.6 Continued

```
            try {
                Properties props = new Properties();
                Session session = Session.getDefaultInstance(props, null);
                Store store = session.getStore("pop3");
                store.connect(host, username, password);
                Folder folder = store.getFolder("INBOX");
                folder.open(Folder.READ_ONLY);
                Message message = folder.getMessage(1);
                Multipart multipart = (Multipart)message.getContent();

                // Process each part of the message
                for (int i=0; i<multipart.getCount(); i++) {
                    processPart(multipart.getBodyPart(i));
                }

                folder.close(false);
                store.close();
            }
            catch (MessagingException me) {
                 System.err.println(me.getMessage());
            }
            catch (IOException ioe) {
                System.err.println(ioe.getMessage());
            }
        }

    private static void processPart(Part part)
    ➥throws MessagingException, IOException{
            String disposition = part.getDisposition();
            String fileName = part.getFileName();

            if (disposition.equals(Part.ATTACHMENT)) {
                // It's an attachment
                if (fileName == null) {
                    // the file name is null, so assign a name
                    fileName = File.createTempFile("attachment", ".txt").getName();
                }
                // write the part to a file
                writeFile(fileName,part.getInputStream());
            }
            else {
                // the disposition is either INLINE or null
            }
        }

    private static void writeFile
    ➥(String fileName, InputStream in) throws IOException {
            // Do no overwrite existing file
            File file = new File(fileName);
```

LISTING 11.6 Continued

```
        for (int i=0; file.exists(); i++) {
            file = new File(fileName+i);
        }
        // Write the part to file
BufferedOutputStream bos =
➥new BufferedOutputStream(new FileOutputStream(file));
        BufferedInputStream bis = new BufferedInputStream(in);
        int aByte;
        while ((aByte = bis.read()) != -1) {
            bos.write(aByte);
        }
        bos.flush();
        bos.close();
        bis.close();
    }
}
```

Authenticating Users and Security

The JavaMail API `javax.mail` package provides an `Authenticator` class that allows access to protected resources, such as a mail box. If you are familiar with the `java.net` package, you may be aware of another class with the same name. However, the two classes are different.

This next application is based on the `RetrieveMail` application you wrote earlier, so this section only walks through the code that differs from that application. You can still find a full listing at the end of this section (Listing 11.7) or on the CD-ROM that accompanies this book. Previously, you passed the host, username, and password to the `Store` object's `connect()` method. In contrast, this application will prompt the user for a username and password, and the host is placed as a system property:

```
Properties props = new Properties();
props.put("mail.pop3.host",host);
```

You then create a new `Authenticator` object by using the `MyAuthenticator` class. This class is a subclass of `Authenticator`, which itself is abstract. You will write the `MyAuthenticator` class shortly. After you create an `Authenticator` object, you pass it together with the `Properties` object to the `getDefaultInstance()` method of the `Session` class:

```
Authenticator auth = new MyAuthenticator();
Session session = Session.getDefaultInstance(props, auth);
```

After you have a `Session` object, you create a `Store` object in the same way you did in the previous applications. You then connect to the store using its `connect()` method, but

in this example, you pass no parameters—previously you passed the `host`, `username`, and `password`. The reason for this is that you have already supplied the information to the `Session` object.

```
Store store = session.getStore("pop3");
store.connect();
```

That is the extent of the changes to the application's main class. Now you must write the `MyAuthenticator` class. This class must define one method, `getPasswordAuthentication()`, and return a `PasswordAuthentication` object. This object is simply a container for the authentication information and has just one constructor that accepts two strings—a username and a password. The method body in this example, which you must implement, simply prompts the user for his or her username and password.

> As mentioned previously, the JavaMail does not directly support secure mail protocols, but some third-parties do provide this support. As such, it is important to remember that the `PasswordAuthentication` object acts only as a container for credential information, and it does not perform any form of encryption on the information.


```
class MyAuthenticator extends Authenticator {
    public PasswordAuthentication getPasswordAuthentication() {
        String username=null;
        String password=null;
        try {
BufferedReader in =
➥new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Username? ");
            username=in.readLine();
            System.out.print("Password? ");
            password=in.readLine();
        }
        catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
        return new PasswordAuthentication(username,password);
    }
}
```

That's it. Listing 11.7 shows the complete code for this application. To run it, compile the code and then issue the following command:

```
java AuthenticateRetrieveMail mail.host
```

LISTING 11.7 AuthenticateRetrieveMail.java Full Listing

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class AuthenticateRetrieveMail {
    public static void main(String[] args) {
        if (args.length!=1) {
            System.out.println("Usage: AuthenticateRetrieveMail SMTPHost");
            System.exit(1);
        }

        String host = args[0];

        try {
            Properties props = new Properties();

            //place the authentication info in
            props.put("mail.pop3.host",host);

            // create an empty authenticator object
            Authenticator auth = new MyAuthenticator();

            // Get a session - pass auth object
            Session session = Session.getDefaultInstance(props, auth);

            Store store = session.getStore("pop3");

            // do not pass any arguments to the connect method
            store.connect();

           Folder folder = store.getFolder("INBOX");
           folder.open(Folder.READ_ONLY);
           Message messages[] = folder.getMessages();
           for (int i=0; i<messages.length; i++) {
System.out.println(i + ": " + messages[i].getFrom()[0]
➥+ "\t" + messages[i].getSubject() + "\t"
➥+ messages[i].getSentDate() + "\n\n");
                messages[i].writeTo(System.out);
            }
            folder.close(false);
            store.close();
        }
        catch (MessagingException me) {
            System.err.println(me.getMessage());
        }
        catch (IOException ioe) {
             System.err.println(ioe.getMessage());
```

LISTING 11.7 Continued

```
        }
    }
}

class MyAuthenticator extends Authenticator {
    public PasswordAuthentication getPasswordAuthentication() {
        String username=null;
        String password=null;
        try {
BufferedReader in =
➥new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Username? ");
            username=in.readLine();
            System.out.print("Password? ");
           password=in.readLine();
        }
        catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
        return new PasswordAuthentication(username,password);
    }
}
```


Summary

In today's lesson, you learned all about e-mail systems and the JavaMail API. The lesson introduced you to the main packages and classes of the API. Building on this, you saw how to write code that performed a number of day-to-day e-mail tasks, such as sending, retrieving and deleting messages, authenticating users, and sending and retrieving attachments.

Although today's example applications ran from the command line, they are all simple to migrate to fit within J2EE components, such as EJBs, servlets, and JSPs. In addition, you can quite simply upgrade the user interfaces for these applications so that they used Swing or AWT components.

Q&A

Q Why does an e-mail system require retrieval protocols?

A When a mail server receives a message, it places it within the recipient's mail box. The recipient then connects to the mail server and downloads the message to his or

her local machine. The sequence of exchanges between the recipient's machine and the mail server use a retrieval protocol, such as POP3 or IMAP.

Q Why do I need to use the Message class's inner class `RecipientType` when adding a recipient to a message?

A There are three types of recipient, TO, CC, and BCC, so you must stipulate the type of recipient you want to add to a message. The `RecipientType` class exposes three static fields that represent the three types of recipients.

Q I want to send a HTML message that includes an embedded image. Why do I need to create a multi-part MIME message?

A MIME supports different data formats. It allows you to mix these formats, to create an inline message, by defining messages that consist of multiple body parts. In the case of a HTML message with inline image, one body part contains the HTML and the other part contains the image. The e-mail client will then assemble these parts to create a compound message.

Q Which method would I use to print the content of messages retrieved from a mail server?

A The `Part` interface's `writeTo()` method allows you to output a bytestream for a message part. However, any class that uses the method must provide an appropriate encoding algorithm because the method does not decode a part's content.

Q I've written an application that checks whether messages are flagged as `RECENT`. If they are, the application retrieves them from the server. Why does the application not retrieve any messages from my POP3 server?

A The JavaMail API provides a selection of predefined message flags, but there is no guarantee that a mail server will support all these flags. Typically, in the case of a POP3 server, only the `DELETED` flag is supported, so in this instance, the application will not find any messages marked as `RECENT`, and thus will not return any messages.

Q Why does the MIME Content-Disposition message header mark some messages as inline and others as attached?

A The Content-Disposition header has two possible values—inline and attached. An application should use these values to determine how to display a body part's content. Specifically, parts marked inline should display within the message, and those marked attached should be saved to file because they are attachments.

Q I'm writing a servlet that retrieves a user's message from an IMAP server. Each user has a unique mail box on the server, how do I ensure that users only access their personal mail boxes?

A The `Authenticator` class allows access to protected resources. You can subclass it, and this subclass can prompt the user for a username and password.

Exercises

1. Write an application that executes from the command line and allows a user to send an e-mail. The application should prompt the user for his or her mail host and e-mail address, the recipient's e-mail address, the message subject, and the message content.
2. There are two parts to this final exercise, and both relate to the case study application you have been building throughout this book:
    - When a customer registers a new job, a message is sent to the `Job Match` Message-driven bean. The bean then updates the `Matched` table with applicants that match that job. Now, write code using the JavaMail API that sends an e-mail message to the customer informing him or her of all the applicants that match his or her job specification.
    - When a new applicant registers, a message is sent to the `Applicant Match` Message-driven bean. The bean updates the `Matched` table with jobs that match that applicant. Now, write code using the JavaMail API that sends an e-mail message to the applicant informing him or her of the jobs to which he or she is suited.

WEEK 2

DAY 12

Servlets

On Day 11, "JavaMail," you worked on integrating JavaMail into your applications. This built on the previous chapters that covered the Java Message System and Message-driven Beans, and you have now completed this book's coverage of asynchronous messaging in J2EE.

Today, you will start to work on the new topic area—providing a Web interface to your application. You will start by using Java servlets with a Web server to handle HTTP requests. These servlets will generate HTML responses to be displayed by using a browser. In today's lesson, you will learn

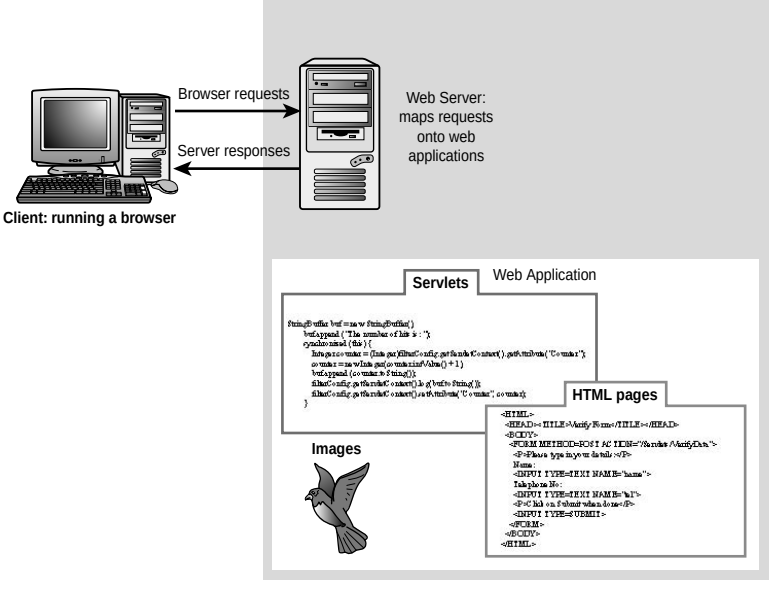
- The power of Java servlets and when to use them
- About the supporting technologies HTTP and HTML
- How to create and track HTTP sessions
- How to develop a Web application using servlets, with servlet filtering and event listening

The Purpose and Use of Servlets

A servlet is a server-side component. It can be used to extend the functionality of any Java-enabled server, but most commonly servlets are used to write Web applications in a Web server, as shown in Figure 12.1. They are often used to create Web pages where the content is not static. Web pages whose content can change according to input from the user or other variable data are called *dynamic pages*. Servlets are particularly suited to creating dynamic Web pages.

FIGURE 12.1

Client/server diagram showing servlets.



The following are the key features and benefits of Java servlets:

- The servlet API provides an interface that is tailored for Web applications.
- Servlets are server and platform independent. This makes servlets portable and reusable.
- Servlets are efficient and scalable.
- Servlets run within the server, so they can delegate certain functions to be performed by the server on its behalf, such as user authentication.

Tailored for Web Applications

A servlet is an instance of a class that implements the `javax.servlet.Servlet` interface. However, most servlets extend one of the implementations of this interface—`javax.servlet.GenericServlet` or `javax.servlet.http.HttpServlet`.

The `Servlet` interface declares methods that manage the servlet and its communications with clients. As the servlet developer, you override some or all of these methods to develop your servlet.

Generic servlets have a limited use, so in today's lesson, we will only discuss the more useful `HttpServlet` class. This is an abstract class that is sub-classed to create an HTTP servlet suitable for a Web site. To accomplish this, an HTTP servlet has access to a library of HTTP-specific calls.

Server and Platform Independence

Java servlets are highly portable between different operating systems and server implementations. A servlet written on a Windows-based PC running the J2EE RI can be deployed on a high-end Unix server without any change at all. For this reason servlets have been described as "write once, serve everywhere."

Servlets have no *client interface*. That means they avoid all the portability issues associated with different display interfaces. An application on the client (typically a browser) takes care of the user interface on behalf of the servlet.

Efficient and Scalable

After being loaded, a servlet will generally stay resident in the server's memory. In most circumstances, only a single servlet object will be created, and to support concurrent page accesses this servlet is run multi-threaded. This avoids the overhead of constructing a new servlet process for every access, saves memory, and makes page access efficient.

Because servlets stay in memory, they can retain references to other Java objects.

For example, if your database server includes sufficient simultaneous connection licenses, a database connection can be shared between threads, thereby reducing the overhead associated with establishing and maintaining the connection.

Multithreading aids efficiency and scalability, but the servlet code must be written to be re-entrant. This means that the servlet must handle concurrent access to instance data, and care must be taken to synchronize write access to shared resources.

Servlets Integration with the Server

Because a servlet is tightly integrated with the server, it can utilize capabilities of the server to perform certain actions. It can, for example, use the server's logging capabilities and get the server to authenticate users.

It may not be possible to take advantage of all server capabilities if you want your servlet to be portable to other platforms and environments.

Although useful for the servlet programmer, the tight coupling of servlets has a safety implication for the Web server. To protect itself, the server will often run servlets in a controlled environment, called a *sandbox*, that is designed to protect the server from a malicious or poorly written servlet.

Introduction to HTTP

Before looking at Java servlets in more detail, you will need an understanding of the Web protocol, HTTP (Hypertext Transfer Protocol), and how a browser interprets HTML (Hypertext Markup Language) to display a Web page. If you are comfortable with these topics, feel free to skip to the next section, titled "The Servlet Environment."

HTTP is a protocol standard specified by the Internet Engineering Task force, and its current version is available as RFC 2616 available from `www.ietf.org`.

HTTP Structure

HTTP is a stateless protocol, and this has the advantage that the server does not have the overhead of tracking client connections. This is completely satisfactory when the primary use of the Web is to transfer static data. Realistically, most Web applications now require interaction between the client and the server and state information to be retained between page requests. Later, you will learn how a servlet can overcome this restriction with the HTTP protocol by tracking client state using hidden fields or cookies.

HTTP transactions are either a *request* or a *response*. Regardless of which type it is, all HTTP transactions have three parts:

- *A single request or response line*—A client request line consists of an HTTP method (usually `GET` or `POST`) followed by a document address and the HTTP version number being used. For example,

  ```
  GET /contents.html HTTP/1.1
  ```

  uses the HTTP `GET` method to request the document `contents.html` using HTTP version 1.1. The response line contains a HTTP status code that indicates whether the request was successful (understood and satisfied) or if not, why not.

- *The HTTP headers*—This is a set of fields used to exchange information between the client and the server. For example, the following tells the server that the client will accept the IOS8859.5 and unicode1.1 character sets:

  ```
  Accept-Charset: iso-8859-5, unicode-1-1
  ```

  The client uses the headers to tell the server about its configuration and the document types it will accept. The server, in turn, uses the header to return information about the requested document, such as its age and location.
- *The HTTP body*—The HTTP body is optionally used by the client to send any additional information (see `POST` method). The server uses the body to return the requested document.

Listing 12.1 shows an example `GET` request. A `GET` request does not have a body, so there are only the request line and headers in this example.

LISTING 12.1 An Example HTTP `GET` Request

```
   1: GET /some/url.html HTTP/1.1
   2: Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
➥application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint,
*/*
   3: Referer: http://www.somewhere.com/search?sourceid=navclient&
➥q=http+request+
   4: Accept-Language: en-gb
   5: Accept-Encoding: gzip
   6: User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
   7: Host: localhost:8000
   8: Connection: Keep-Alive
```


See RFC 2616 (available from `www.ietf.org`) for the meaning of these header fields and for more information on HTTP in general.

Uniform Resource Identifiers (URIs)

A Universal Resource Identifier (URI) is also commonly known as a Universal Resource Locator (URL). URIs are used on the Web to identify documents, images, downloadable files, services, electronic mailboxes, and other resources.

Using a simple syntax, URIs make use of a variety of naming schemes and access methods, such as HTTP, FTP, and Internet mail to identify online resources.

The syntax of an HTTP URL is as follows:

```
http_URL = "http://" host [ ":" port] [ path ]
```

where

- *host* is a legal Internet host domain name or IP address (in dotted-decimal form).
- *port* is the port number (also known as the socket or service number) to connect to on the host. The default port number is 80—the TCP/IP port.
- *path* is the path to the document on the host.

The term URL is more commonly used when referring to the HTTP address string and, for this reason, URL is the term that will be used for the rest of today's material.

Space characters should be avoided in URL's because they may not be handled correctly on all platforms.

HTTP GET and POST Methods

A Web browser client communicates with the server typically using one of two HTTP methods—GET or POST. Typically, these methods are used as follows:

- GET is used to request information from the server.
- POST is used to send data to the server.

But as with many things, it is not quite that simple. The GET method can also be used to pass information in the form of a query string in the URL, and POST can be used for requests.

The following URL with a query string (the data following the ?) is passed by the GET method and sets a parameter called day to the value 12 (you will learn more about parameters later when you code some real servlets).

```
http://localhost:8000/j2ee?day=12
```

Because the query string is added to the end of the URL, information that is sent as part of a GET request is visible to the client. You will have seen examples of this many times when browsing the Web, especially when using search engines.

Because a URL can be read and bookmarked, sensitive information (such as a password, personal details, or credit card numbers) should not be passed this way.

In contrast, the POST method sends its data directly after the HTTP header, in the body of the message, and does not append data to the URL. For this reason, it is safer to send sensitive information using POST.

In all other respects, the GET and POST methods can be thought of as the same. They both interact with the server and can be used to update or change the current Web page and change server-side properties. In fact, POST methods are often used to send long query strings that would otherwise overflow the maximum size for a URL. As a servlet developer, whether to use GET or POST can become a matter of personal choice, as long as you remember that GET should never be used where the information being sent may be used to compromise someone's security or privacy.

Other HTTP Methods

The following HTTP methods are used less often, but are covered here for completeness.

- HEAD This method can be used if the client wants information about a document but does not want the document to be returned. Following a HEAD request, the server responds with the HTTP headers only; no HTTP body is sent.
- PUT Requests the server to store the body of the request at a specified URL.
- DELETE Requests the removal of data at a URL.
- OPTIONS Requests information about the communications options available.
- TRACE Used for debugging. The HTTP body is simply returned by the server.

Server Responses

The server sends back a HTTP response to the client that may look something like Listing 12.2.

LISTING 12.2 HTTP Response

```
1: HTTP/1.1 200 OK
2: Date: Tue, 20 Nov 2001 09:23:44 GMT
3: Server: Netscape-Enterprise/3.5.1G
4: Last-modified: Mon, 12 Nov 2001 15:31:26 GMT
5: Content-type: text/html
6: Content-length: 2048
7: Page-Completion-Status: Normal
```

The server sends back a status code as part of the first line of the response followed by header-fields describing the document. A blank line separates the header from the document itself.

Most of the time, the status code is handled by the browser, but you will, no doubt, be familiar with one or two that are reported to the end user. In particular, you will have seen the ubiquitous 404 Not Found error that is sent when the server was unable to find the requested URL.

To aid in coding (and debugging) your servlets, it is useful to have a knowledge of the HTTP status codes. Status codes are grouped as shown in Table 12.1.

TABLE 12.1 HTTP Status Code Groups

Code	*Description*
100-199	Information indicating that the request has been received and is being processed.
200-299	Request was successful.
300-399	Further action is required.
400-499	Request is incomplete.
500-599	Server error has occurred.

Most browsers will deal with most of these status codes silently. The handling of status codes is browser specific, but some status codes you may see include those shown in Table 12.2.

TABLE 12.2 HTTP Status Codes

Code	*Error*	*Description*
400	Bad Request	The server detected a syntax error in the request.
401	Unauthorized	The request did not have the correct authorization.
403	Forbidden	The request was denied, reason unknown.
404	Not Found	The document was not found.
500	Internal Server Error	Usually indicates that part of the server (probably your servlet) has crashed.
501	Not Implemented	The server cannot perform the requested action.

Content Type Headers

As part of the response headers, a content type field is used to indicate the format of the data that is being sent in the response. The value for this field is in Multipurpose Internet Mail Extensions (MIME)—also used when attaching documents to e-mail.

Some self-explanatory MIME content types are as follows:

- `text/html`
- `text/plain`
- `image/gif`
- `application/pdf`

The browser can also specify in the request header the MIME types that it will accept.

Introduction to HTML

HTML is the language of the Web. It is used to encode embedded directions (tags) that indicate to a Web browser how to display the contents of a document.

The HTML standard is under the authority of the World Wide Web Consortium. Unlike the HTTP standard that is used consistently across implementations, browser writers have implemented HTML differently, according to their whim, and have added their own proprietary HTML extensions (to the point that different versions of the same browser may handle the same HTML tag differently).

In your servlet code, you are advised to restrict the use of HTML to well-established tags and features. All the HTML covered here will work in all the most popular browsers, although you may find the output may look different in your favorite.

An HTML document has a well-defined structure consisting of required and optional HTML elements.

An HTML element consists of a tag name followed by an optional list of attributes all enclosed in angle brackets (`<...>`). Tag names and attributes are not case sensitive and cannot contain a space, tab, or return character. Most HTML tags come in pairs—a start tag and an end tag. The end tag is the same as the start tag but has a forward slash character preceding the tag name. For example, an HTML document begins with `<HTML>` and ends with `</HTML>`.

Tags are nested. This means that you must end the most recent tag before ending a preceding one. Apart from this restriction, the actual layout is completely free format. An indented layout can be used to aid readability but is not required.

Each HTML document has an optional HEAD and a BODY. The HEAD is where you pass information to the browser about the document; text in the header is not displayed as the content of the document. The BODY includes the information (tags and text) that defines the document's content. A well-formed (if a little basic) HTML document is shown in Listing 12.3, the output displayed in Microsoft's Internet Explorer is shown in Figure 12.2.

LISTING 12.3 A Simple HTML Page

```
1: <HTML>
2:   <HEAD>
3:     <TITLE>My Very First HTML Document</TITLE>
4:   </HEAD>
5:   <BODY>
6:     <H1>Here is a H1 header</H1>
7:     and here is some text - hopefully it looks different from the header
8:   </BODY>
9: </HTML>
```

FIGURE 12.2

Screen shot of a simple HTML page.

My Very First HTML Document - Microsoft Internet Explorer - [Working Offline]
File Edit View Favorites Tools Help
Search Favorites Media
Address F:\First.html
Go Links
Here is a H1 header
and here is some ordinary text – hopefully it looks different from the header
Done
My Computer

All tags have a name, and some tags may also have one or more attributes that are used to add extra information. Attribute values can be case sensitive and should be enclosed in quotes if they include any space or special characters (if in doubt—quote), both single and double quotes can be used.

A little confusingly, a few common HTML tags do not normally come in pairs because the end tag can be omitted. These include <IMG>, which inserts a graphic image, and <BR>, which causes a line break.

Table 12.3 shows a list of HTML tags and attributes that can be used to format a simple HTML document. Only tags from this list are used in today's lesson. This is not a full list of HTML tags, nor does it show any attributes to the tags. For a definitive list, see the latest HTML specification available from www.w3.org.

TABLE 12.3 Summary of Common HTML Tags

TAG	*Description*
<A>	Create a hyperlink (HREF attribute).
<BIG>	Format the text in a bigger typeface.
<BODY>	Enclose the body of the HTML document.
 	Start a new row of text.
<BUTTON>	Create a button element within a <FORM>
<FONT>	Change the size, color, or typeface of text.
<FORM>	Delimit a form. This is used to send user input.
<H*n*>	Header text, where *n* is a number between 1 and 6.
<HEAD>	Encloses the document head.
<HTML>	Used to delimit the entire HTML document.
<IMG>	Insert an image.
<INPUT>	Create buttons or other elements in a <FORM> used to pass information to the server.
<OPTION>	Define a single option within a <SELECT> list, see <SELECT>.
<P>	Start a new paragraph.
<SELECT>	Start the <OPTION> list for a multiple-choice menu.
<TABLE>	Place text in table format.
<TD>	Define the contents of a data cell in a <TABLE>.
<TH>	Define the contents of a header cell in a <TABLE>.
<TR>	Define a row of cells within a <TABLE>.
<!-- -->	Enclose a comment.

Listing 12.4 is an HTML document that illustrates the use of some of these tags. It contains an input form with a button and outputs data in the form of a table.

LISTING 12.4 Simple HTML Form

```
1: <HTML>
2:   <HEAD><TITLE>Simple HTML Form</TITLE></HEAD>
3:   <BODY><FONT FACE=ARIAL COLOR=DARKBLUE>
4:     <H1><FONT SIZE=+3>Color Selector</FONT></H1>
5:     <FORM METHOD=GET ACTION="http://localhost:8000/servlets/processForm">
6:       Please Type in your name here:
7:       <INPUT TYPE=TEXT NAME="name">
8:       <P>Now select the color of your choice</P>
```

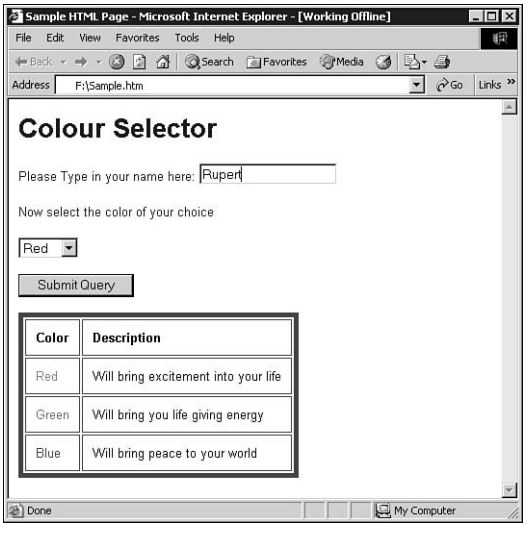
LISTING 12.4 Continued

```
 9:       <SELECT NAME="color" SIZE=1><OPTION>Red<OPTION>Green
➥<OPTION>Blue</SELECT>
10:       <BR><BR>
11:       <INPUT TYPE=SUBMIT>
12:       <BR><BR>
13:       <TABLE BORDER=5 BORDERCOLOR=BLUE CELLPADDING=10>
14:         <TR>
15:           <TH ALIGN=LEFT>Color</TH><TH ALIGN=LEFT>Description</TH>
16:         </TR>
17:         <TR>
18:           <TD><FONT COLOR=RED>Red</FONT></TD>
19:           <TD>Will bring excitement into your life</TD>
20:         </TR>
21:         <TR>
22:           <TD><FONT COLOR=GREEN>Green</FONT></TD>
23:           <TD>Will bring you life giving energy</TD>
24:         </TR>
25:         <TR>
26:           <TD><FONT COLOR=BLUE>Blue</FONT></TD>
27:           <TD>Will bring peace to your world</TD>
28:         </TR>
29:       </TABLE>
30:     </FORM>
31:   </BODY>
32: </HTML>
```

The output of this code in Microsoft's Internet Explorer version 6 is shown in Figure 12.3. The page will look similar, but not necessarily exactly the same, in other browsers.

FIGURE 12.3

Sample HTML page displayed using Microsoft's Internet Explorer version 6.



This completes the discussion of HTML; access the latest standard and other documents on the WC3 Web site (`www.w3.org`) for more information on the standard.

The Servlet Environment

We now turn our attention to servlets. Servlets are Java classes that can be loaded dynamically and run by a Java-enabled Web server. The Web server provides support for servlets with extensions called *containers* (also known as *servlet engines*).

Web clients (Web browsers) interact with the servlet using the HTTP request/response protocol that has been described earlier.

Servlet Containers

The servlet container provides the following services and functionality:

- The network services over which the requests and responses are sent.
- Registers the servlet against one or more URLs.
- Manages the servlet lifecycle.
- Decodes MIME-based requests.
- Constructs MIME-based responses.
- Supports the HTTP protocol (it can also support other protocols).

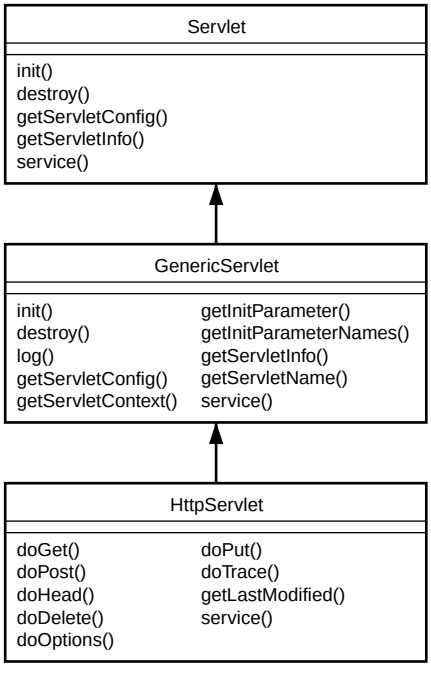
A servlet container can also enforce security restrictions on the environment, such as requesting the user to log in to access a Web page.

The Servlet Class Hierarchy

The Servlet API specification is produced by Sun Microsystems Inc., and a copy of the latest specification can be found on their Web site (`developer.java.sun.com`).

An HTTP servlet extends the `javax.servlet.HttpServlet` class, which itself extends `javax.servlet.GenericServlet`, as shown in Figure 12.4.

FIGURE 12.4
Servlet class hierarchy diagram.



Simple Servlet Example

You are now in a position to write your first servlet.

In Listing 12.5, you will generate a complete HTML page that displays a simple text string. This servlet extends the `HttpServlet` class and overrides the `doGet()` method.

LISTING 12.5 A Servlet Generating a Complete HTML Page

```
   1: import java.io.*;
   2: import javax.servlet.*;
   3: import javax.servlet.http.*;
   4:
   5: public class HTMLPage extends HttpServlet {
   6:     public void doGet(HttpServletRequest req, HttpServletResponse res)
➥throws IOException {
   7:         res.setContentType ("text/html");    // the content's MIME type
   8:         PrintWriter out = res.getWriter();   // access the output stream
   9:         out.println ("<HTML>");
  10:         out.println ("<HEAD><TITLE>First Servlet</TITLE></HEAD>");
  11:         out.println ("<BODY>");
  12:         out.println ("<H1>My First Servlet Generated HTML Page</H1>");
  13:         out.println ("</BODY>");
```

LISTING 12.5 Continued

```
14:         out.println ("</HTML>");
15:     }
16: }
```

When the client browser sends the `GET` request for this servlet, the server invokes your `doGet()` method, passing it the HTTP request in the `HttPServletRequest` object. The response is sent back to the client in the `HttpServletResponse` object.

The `HttPServletRequest` interface provides access to information about the request, and its main use is to give access to parameters passed in the URL query string. For this simple example, the request object does not contain any useful information, so it is not accessed in this servlet. You will see how to use the `HttPServletRequest` object in later examples.

The `HttpServletResponse` object is used to return data to the client. For this simple example, the data is sent as the MIME type `text/html`. The `PrintWriter` object encodes the data that is sent to the client.

A call to the `setContentType()` method should always be made before obtaining a `PrintWriter` object.

Lines 9–14 in Listing 12.5 generate the HTML that the browser will use to display the page.

After compiling the code, start up the J2EE RI and run `deploytool`. You will now perform the following steps to deploy the servlet.

1. Create a new application called **`Servlets`** to store your servlet code.
2. From the File menu, select New, Web Component.
3. On the first input page, choose Create New WAR File in Application. Click the Edit button and add the **`HTMLPage`** class file (it will put the class file in a `WEB-INF` directory, why is explained later), as shown in Figure 12.5.
4. On the next page, choose Web component type to be Servlet, as shown in Figure 12.6.

FIGURE 12.5

`deploytool` WAR page.

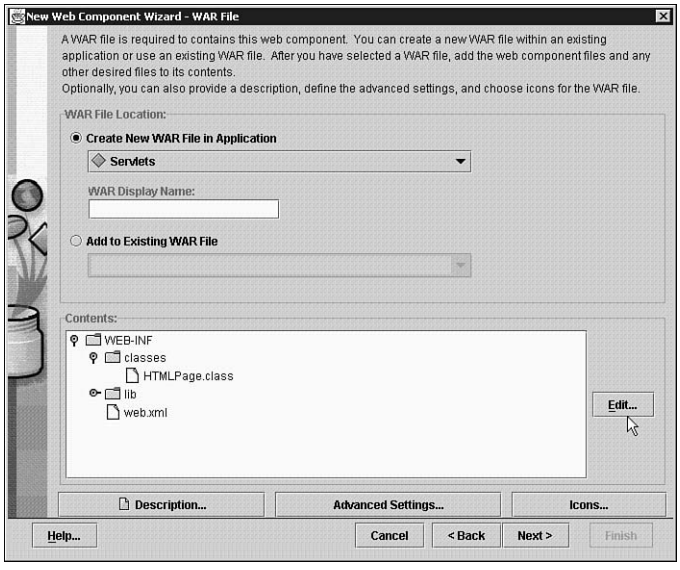
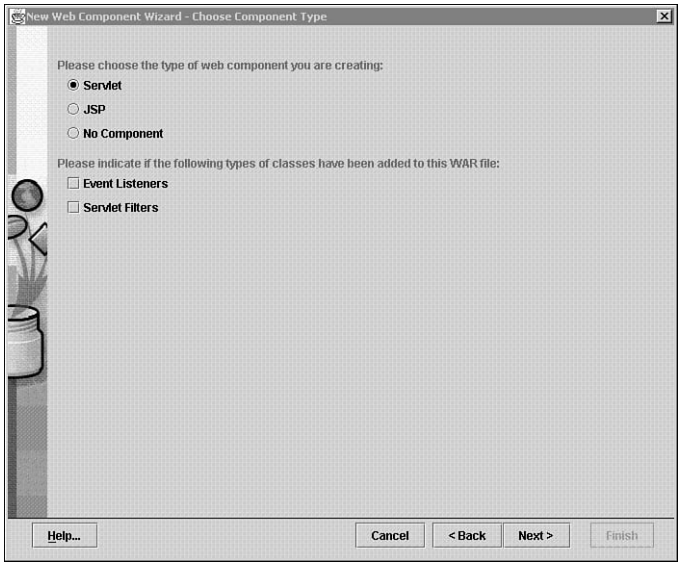

New Web Component Wizard - WAR File
A WAR file is required to contains this web component. You can create a new WAR file within an existing application or use an existing WAR file. After you have selected a WAR file, add the web component files and any other desired files to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.
WAR File Location:
Create New WAR File in Application
Servlets
WAR Display Name:
Add to Existing WAR File
Contents:
WEB-INF
classes
HTMLPage.class
lib
web.xml
Edit...
Description...
Advanced Settings...
Icons...
Help...
Cancel
< Back
Next >
Finish

FIGURE 12.6

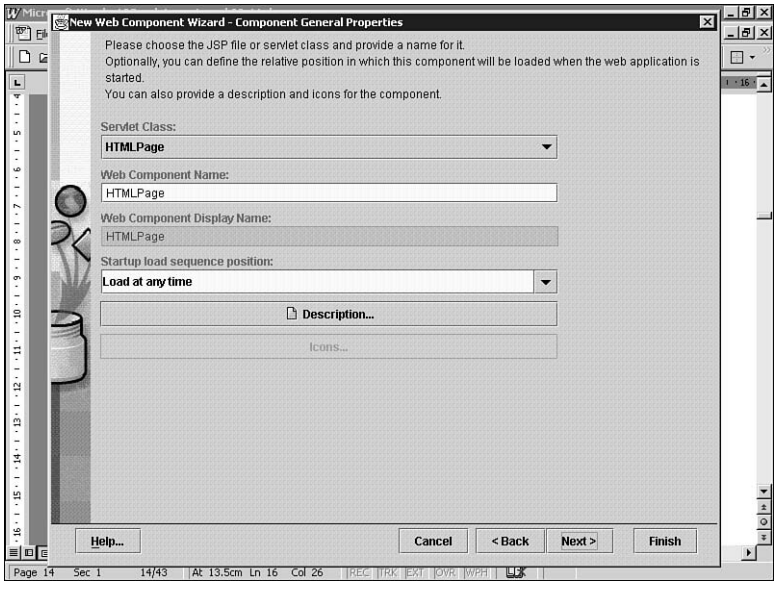
`deploytool` Choose Component Type page.


New Web Component Wizard - Choose Component Type
Please choose the type of web component you are creating:
Servlet
JSP
No Component
Please indicate if the following types of classes have been added to this WAR file:
Event Listeners
Servlet Filters
Help...
Cancel
< Back
Next >
Finish

5. On the next page, select the **`HTMLPage`** as the Servlet Class (see Figure 12.7). Take the defaults for the remainder of the parameters on this screen.

FIGURE 12.7

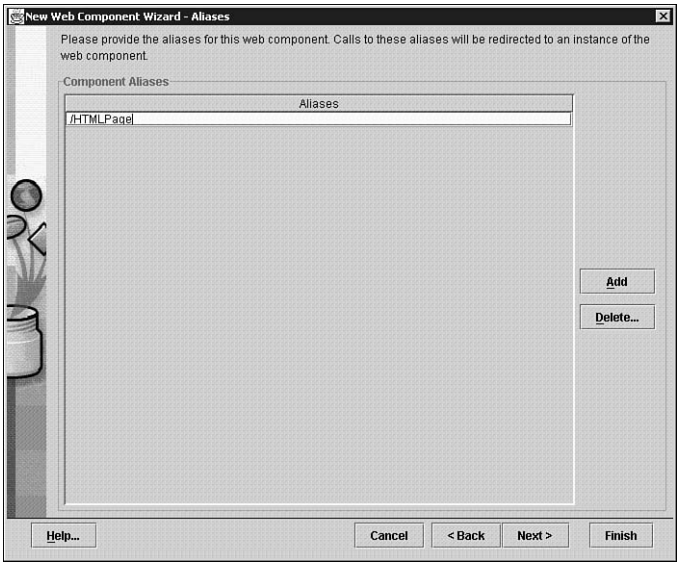
`deploytool` Component General Properties page.


New Web Component Wizard - Component General Properties
Please choose the JSP file or servlet class and provide a name for it.
Optionally, you can define the relative position in which this component will be loaded when the web application is started.
You can also provide a description and icons for the component.
Servlet Class:
HTMLPage
Web Component Name:
HTMLPage
Web Component Display Name:
HTMLPage
Startup load sequence position:
Load at any time
Description...
Icons...
Help...
Cancel
< Back
Next >
Finish

6. There are no initialization parameters, so skip the next page.
7. On the next page, New Web Component Wizard—Aliases, add an alias for this page. Use the same name as the servlet and precede it with a `/` (**`/HTMLPage`**), as shown in Figure 12.8. The component alias is very important; it is used in the URL used to access the page. Without it, you will not be able to access your servlet.

FIGURE 12.8

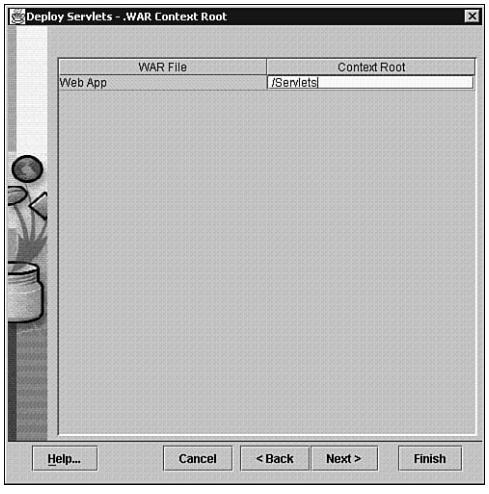
`deploytool` Aliases page.



8. There is no further configuration to do, so select Finish.
9. As always, select the Tools, Verifier menu to check the application before deployment. Ignore any warning about a missing context because you will add this in the next step.
10. To deploy the application, you will need to supply a context root; use `/Servlets` (see Figure 12.9).

FIGURE 12.9

`deploytool` .WAR Context Root page.



The context root is used in the URL to find the WAR file. The final URL will be

```
http://<Web server address>/<Context root>/<Component alias>
```

For this example, if you have used the names suggested, the URL is as follows:

```
http://localhost:8000/Servlets/HTMLPage
```

Now, start up a Web browser. First, type in the URL for the Web server (`http://localhost:8000`) and check that server is up and running (see Figure 12.10).

If you have used the Web server before, you may not see this page. If you have problems starting the Web server, Sun Microsystems has provided an e-mail address (`J2EE-ri-feedback@sun.com`), but this is not a commercial application and a response is not guaranteed.

Now access your servlet using the following URL (see Figure 12.11):

```
http://localhost:8000/Servlets/HTMLPage.
```

FIGURE 12.10
J2EE 1.3 server page.

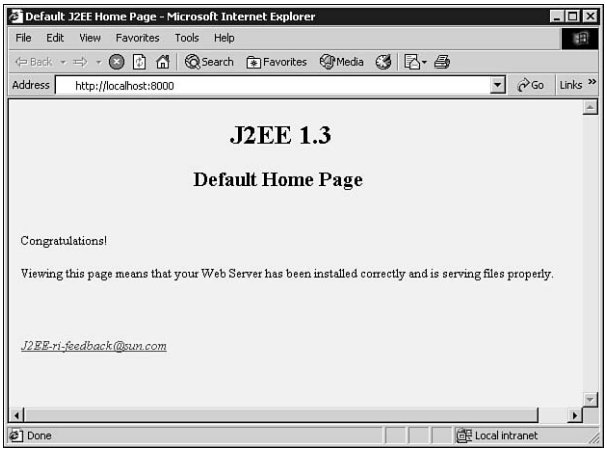

Default J2EE Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Search Favorites Media
Address http://localhost:8000
Go Links
J2EE 1.3
Default Home Page
Congratulations!
Viewing this page means that your Web Server has been installed correctly and is serving files properly.
J2EE-ri-feedback@sun.com
Done
Local intranet

FIGURE 12.11
HTMLPage servlet.

Congratulations, you have just produced your first servlet.

Passing Parameter Data to a Servlet

Your first servlet, although it shows the principle of servlets, was not actually a very good servlet example. As you have probably realized, it could have been written as a static HTML page and you could have avoided all the complication of having to compile and deploy a servlet. The next example will show the power of servlets.

How to Access Parameters

The power of servlets comes into its own when the data is dynamic. Dynamic data can come from many sources. It can be obtained from an external resource (such as a file, database, or another Web page) or it can be sent from a HTML form to the server in the form of parameters.

Parameters are name-value pairs. Parameters are defined in the HTML, for example

```
<INPUT TYPE=TEXT NAME="myname">
```

places an input text box on the page and any data typed in the input box is associated with a parameter called `myname`.

If the form uses a `GET` method, this parameter will be added to the URL as part of the query string, as in the following:

```
http://localhost:8000/Servlets/example?myname=Rupert
```

In the servlet, you use the `HttpServletRequest.getParameter()` method to gain access to parameters. For example, the following line will obtain the value stored in the `myname` parameter:

```
String name = req.getParameter("myname");
```

Servlet Example with Parameters

Youwill now code an application takes information from a simple form and displays it back to the user. There are two parts to the application:

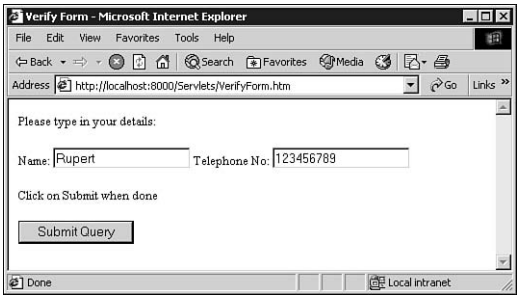
- `VerifyForm` A static HTML form that obtains the data from the user (shown in Listing 12.6).
- `VerifyData` A servlet that simply sends the entered data back to the client (shown in Listing 12.7). This is actually quite a common requirement, where the user is presented with the data just typed into a form to visually verify it before the application continues processing it.

If you deploy the form, `VerifyForm`, as part of your Web application, you can use a relative path to access the servlet `"/Servlets/VerifyData"` (line 4 of Listing 12.6). Otherwise, you will need the full path `"http://localhost:8000/Servlets/VerifyData"`.

LISTING 12.6 `VerifyForm` HTML Page to Obtain the Data

```
 1: <HTML>
 2:   <HEAD><TITLE>Verify Form</TITLE></HEAD>
 3:   <BODY>
 4:     <FORM METHOD=GET ACTION="/Servlets/VerifyData">
 5:       <P>Please type in your details:</P>
 6:       Name:
 7:       <INPUT TYPE=TEXT NAME="name">
 8:       Telephone No:
 9:       <INPUT TYPE=TEXT NAME="tel">
10:       <P>Click on Submit when done</P>
11:       <INPUT TYPE=SUBMIT>
12:     </FORM>
13:   </BODY>
14: </HTML>
```

FIGURE 12.12
VerifyForm page.



When you click the Submit Query button, the text typed in the two input boxes is sent as a query string to the servlet. The query string is appended at the end of the URL that was specified as the `ACTION` attribute (line 4 in Listing 12.6) to the form, as shown in the following:

```
http://localhost:8000/Servlets/VerifyData?name=Rupert+&tel=123456789
```

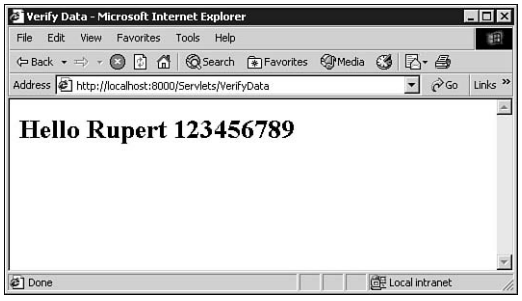
The `VerifyData` servlet (shown in full in Listing 12.7) uses the `getParameter()` method to access these parameters. Deploy this servlet as part of your Servlets application.

LISTING 12.7 `VerifyData` Servlet with Parameters

```
 1: import java.io.*;
 2: import javax.servlet.*;
 3: import javax.servlet.http.*;
 4:
 5: public class VerifyData extends HttpServlet {
 6:
 7:     public void doGet(HttpServletRequest req, HttpServletResponse res)
 8:                  throws IOException {
 9:         res.setContentType ("text/html");
10:         PrintWriter out = res.getWriter();
11:         String name = req.getParameter("name");
12:         String tel = req.getParameter("tel");
13:         out.println ("<HTML>");
14:         out.println ("<HEAD><TITLE>Verify Data</TITLE></HEAD>");
15:         out.println ("<BODY>");
16:         out.println ("<H1>Hello " + name + " " + tel + "</H1>");
17:         out.println ("</BODY>");
18:         out.println ("</HTML>");
19:     }
20: }
```

If everything is correct and you have typed in the data in Figure 12.12, this servlet produces the output shown in Figure 12.13.

FIGURE 12.13
VerifyData page.



Using a POST Request

As already discussed, both the HTTP methods `POST` and `GET` can be used to send data to a servlet. Generally, you need the functionality to be the same if either `GET` or `POST` is used.

The simplest way for a servlet to handle `POST` requests is to dispatch the request to the `doGet()` method. Add the following `doPost()` method to your servlet:

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException {
    doGet (req, res);
}
```

Now change your form to use `POST` instead of `GET`. To do this, change line 4 of Listing 12.6 to the following:

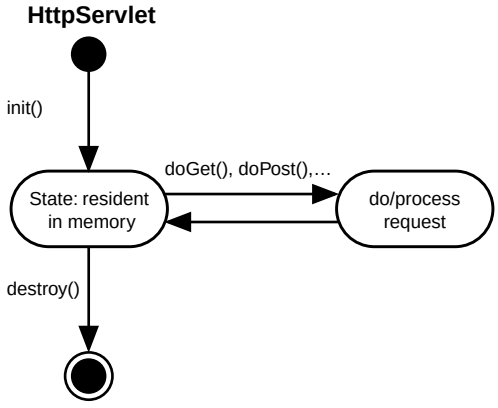
```
<FORM METHOD=POST ACTION="/Servlets/VerifyData">
```

You will notice that the parameters are not included as part of the URL this time, but the output and the effect is exactly the same as when `GET` was used.

The Servlet Lifecycle

The servlet specification defines the lifecycle of a servlet, this is shown in Figure 12.14, and the servlet container's responsibilities. The server is responsible for loading, instantiating, and initializing servlets. But exactly when this happens is not defined, a server may instantiate all servlets when the servlet engine is started, or instantiation of a servlet may be delayed until it is needed. The lifespan of a servlet is also non-deterministic. A server may keep a servlet active for a long time, or the server may remove a servlet when resources are low.

FIGURE 12.14
Lifecycle of a servlet.



When a servlet is instantiated, the server calls the servlet's `init()` method. This is called once and once only, and should be used to set up servlet resources and initialize servlet instance variables. A servlet can access its context in the `init()` method.

The `init()` method must complete before the servlet can service any requests. The servlet will not be put into service if the `init()` method throws a `ServletException`.

The `init()` method is defined in the superclass. You only need override this method if you need to do some initialization for the servlet.

The `destroy()` method is called by the server when the servlet is removed from service. Because a server can destroy a servlet at any time, it should be used not only to free resources but also to save any data or state information that may be required when the servlet is called again. After a servlet has been initialized, request and response objects are passed as parameters to the appropriate `do`*`XXX`*`()` method (see Figure 12.15) of the servlet interface.

These objects are as follows:

- `javax.servlet.http.HttpServletRequest`
- `javax.servlet.http.HttpServletResponse`

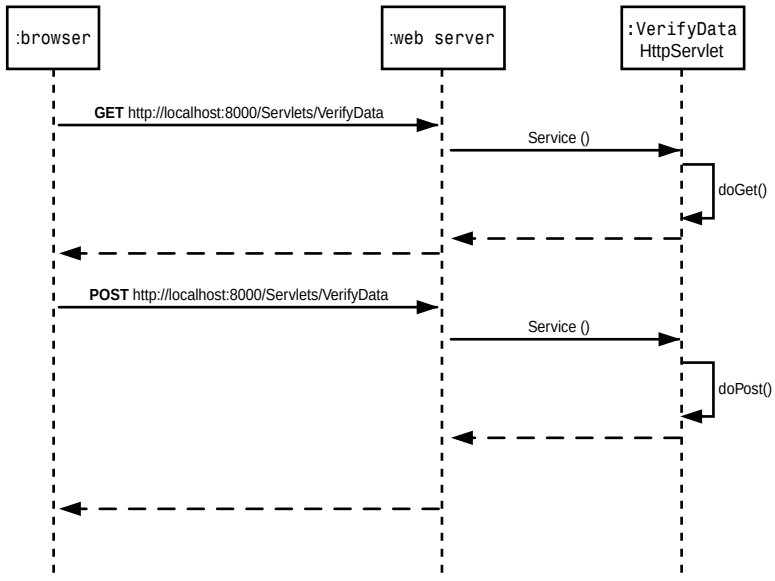


FIGURE 12.15
Sequence diagram of HTTP `GET` and `POST` requests.

The Servlet Context

The `javax.servlet.ServletContext` interface provides a set of methods that the servlet can use to communicate with the Web server. The `ServletContext` object is contained within the `javax.servlet.ServletConfig` object, which is provided to the servlet when it is first initialized.

Using the `ServletContext` object, a servlet can perform the following functions:

- Set and store attributes that other servlets in the context can access.
- Log events.
- Obtain URL references to resources.
- Get values assigned to initialization parameters. These are parameters associated with a servlet, not an individual request.
- Get the MIME type of files.
- Obtain information about the servlet container, such as its name and version.

A servlet context is associated with a Web application and shared by all the servlets within that application.

Web Applications

In this section, Web applications will be described in more detail—what happens when one is created and what is happening behind the scenes when you use the J2EE RI.

Web Application Files and Directory Structure

Web applications are run on a Web server. A Web application is simply a set of static HTML pages, servlets, support classes, applets, JavaBeans, and any other resource that bundled together form a complete application. The J2EE RI packages Web applications for you and stores them in a Web Archive Format (`WAR`) file so they can be easily distributed to other Web servers.

The Java Servlets specification defines a structured hierarchy of directories that are used for deployment and packaging purposes. Using the J2EE RI `deploytool`, class files and other configuration files are placed in the directory structure for you. If you use a different Web server (such as Jakarta Tomcat), you may need to manually create and place files in the correct place.

Each Web application has a root called the context path. You entered the context path when you deployed your servlets. No two applications can have the same context path because this would cause potential URL conflicts.

The context root is usually mapped onto a physical directory called the *document root*. Using the J2EE RI, this directory is found in `/public_html` under the J2EE installation directory. This is also where the `WAR` file is stored, if there is one.

Below the document root is a special directory called `WEB-INF`. The contents of the `WEB-INF` directory are as follows:

- `web.xml` file—The Web application deployment descriptor
- `/classes` directory—Used to store the class files for the servlets and utility classes
- `/lib` directory—Storage area for Java Archive (JAR) files

If the `WAR` file exists, there will also be a `META-INF` directory that contains information useful to Java Archive tools.

For a small application, the directory structure might look like the following:

```
\images\background.gif
\images\cutout.gif
\index.html
\META-INF\MANIFEST.MF
\servlets.war
\VerifyForm.html
```

```
\WEB-INF\classes\HTMLPage.class
\WEB-INF\classes\VerifyData.class
\WEB-INF\lib\bean.jar
\WEB-INF\web.xml
```

The Web Application Deployment Descriptor

Listing 12.8 is the deployment descriptor for a simple Web application showing the major features. This example is based on one given in the Servlets 2.3 specification.

It begins with two tags (`<?xml> <!DOCTYPE>`) that define the XML document structure. The remainder of the XML describes the Web application and sets parameters for the servlets.

LISTING 12.8 Deployment Descriptor for a Simple Web Application

```
   1: <?xml version="1.0" encoding="UTF-8"?>   2: <!DOCTYPE web-app
➥PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
➥'http://java.sun.com/dtd/web-app_2_3.dtd'>   3: <web-app>
   4:   <display-name>A Simple Application</display-name>
   5:   <servlet>
   6:     <servlet-name>catalog</servlet-name>
   7:     <servlet-class>CatalogServlet</servlet-class>
   8:     <init-param>
   9:       <param-name>catalog</param-name>
  10:       <param-value>Spring</param-value>
  11:     </init-param>
  12:   </servlet>
  13:   <servlet-mapping>
  14:     <servlet-name>catalog</servlet-name>
  15:     <url-pattern>/catalog/*</url-pattern>
  16:   </servlet-mapping>
  17:   <session-config>
  18:     <session-timeout>30</session-timeout>
  19:   </session-config>
  20:   <error-page>
  21:     <error-code>404</error-code>
  22:     <location>/error404.html</location>
  23:   </error-page>
  24: </web-app>
```

Table 12.4 contains a short description of the tags used in this deployment descriptor, see the servlets specification for a more complete list.

TABLE 12.4 XML Tags Used in Listing 12.8

XML Tag	*Description*
`DOCTYPE`	Indicates that this is a deployment descriptor for version 2.3 of the servlet's specification.
`web-app`	Root for the deployment descriptor
`display-name`	Short name for application, need not be unique.
`servlet-name`	The official name of the servlet, must be unique.
`servlet-class`	The servlet's fully-qualified classname.
`init-param`	Initialization parameters available to the servlet. This is followed by name/value pairs of parameters.
`param-name`	The name of the parameter.
`param-value`	The value for the parameter.
`servlet-mapping`	Used to map a servlet to a URL.
`session-config`	Defines timeout behavior for sessions.
`error-page`	Used to map an HTTP error status code onto a Web resource, such as an HTML page, that will be displayed instead of the standard browser error pages (see the "Handling Errors" section later in the chapter).

If you use the J2EE RI, the deployment descriptor is created for you. If you are using other Web servers, you may need to manually create the XML for the deployment descriptor and save it as the `web.xml` file.

To be validated, the deployment descriptor requires a Document Type Definition (DTD) file.

The DTD contains the rules for processing the deployment descriptor. If you are comfortable with the syntax used in DTDs, the deployment descriptor DTD itself can be found by following the URL given in the `DOCTYPE` tag (`http://java.sun.com/dtd/web-app_2_3.dtd`). Also see Appendix C, "An Overview of XML," on the CD-ROM.

For more information on DTDs, see also the Servlet Specification for a full description of the deployment descriptor DTD file.

If you are writing your own deployment descriptor, you will need to follow the DTD rules to ensure that it is valid; otherwise, your application will not be deployed.

Handling Errors

There are a number of possible error or failure conditions that need to be handled by your servlet. These errors fall into the following two categories:

- HTTP errors
- Servlet exceptions

HTTP Errors

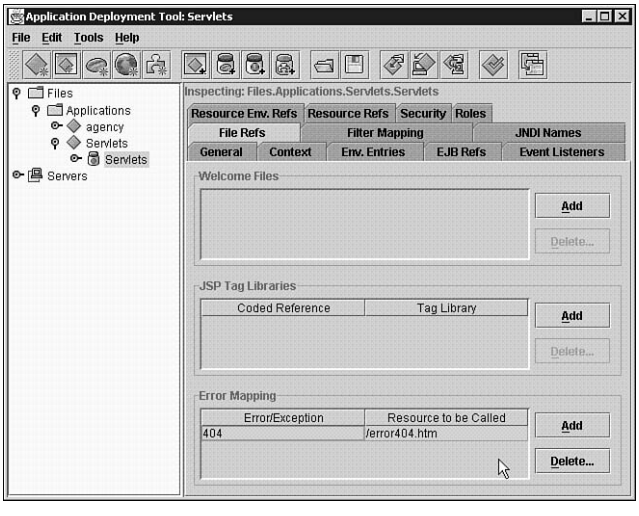
One way of handling HTTP errors in the deployment descriptor has already been briefly mentioned. By using the XML `error-page` tag, you can ensure that the client is sent an application-specific page when it gets an error. You can use this page to send appropriate information in response to any HTTP error code. The following, for example, replaces the standard HTTP "404 Not found" error page with one that will have been written for this application:

```
<error-page>
  <error-code>404</error-code>
  <location>/Servlets/error404.html</location>
</error-page>
```

If the user tries to access a page that is not found on the server, the `/Servlets/error404.html` page will be returned to the client. Using the J2EE RI, this error page redirection can be included in the deployment descriptor by adding an Error Mapping entry to the File Refs tab for your application (see Figure 12.16).

FIGURE 12.16

`deploytool` File Refs tab.



Generating HTTP Status Codes

There may be times when it is useful for the servlet to generate its own HTTP status codes. In an error situation, the servlet can use either of the following methods to set the HTTP status code

```
public void HttpServletResponse.sendError(int sc)
public void HttpServletResponse.sendError(int sc, String msg)
```

The API defines a set of constants that can be used to refer to HTTP status codes. For example, the error status returned can be set to 404, as in the following:

```
res.sendError(res.SC_NOT_FOUND);
```

This will he handled by the server and browser in exactly the same way as any "404 Not found" error, including using any error-page redirection specified in the deployment descriptor.

Send Redirect

Another useful thing a servlet can do if an error is detected is to redirect the client to another URL. This can also be used in obsolete servlets to redirect the client seamlessly to a new application.

Use `HttpServletResponse.sendRedirect(String location)` to redirect the response to the specified location.

The following will redirect the client to another page called `"/Servlets/AnotherHTMLPage"`.

```
res.sendRedirect("/Servlets/AnotherHTMLPage");
```


Servlet Exception Handling

In general, you should take care to catch all servlet-generated exceptions in the servlet and take appropriate action to inform the client what has happened. If you don't, instead of the expected page, the user is likely to see a Java exception stack trace on the screen. Although you may find the information useful during development, an exception stack trace sent to the client will leave most users bewildered and worried.

When a fatal exception occurs, there is often nothing to do but tell the client that a serious error has occurred. The following catch block simply returns a HTTP 503 Service Unavailable error to the client, along with a suitable error message:

```
catch (RemoteException ex) {
    res.sendError(res. SC_SERVICE_UNAVAILABLE, "Internal communication error");
}
```

In other situations, it might be appropriate to redirect the user to another URL that has a form that the user can use to report the error:

```
catch (RemoteException ex) {
    res. sendRedirect("/Servlets/ReportErrorPage");
}
```

A servlet can throw a number of exceptions that will be handled by the server. During initialization or while handling requests, the servlet instance can throw an `UnavailableException` or a `ServletException`. If this happens, the action taken by the server is implementation specific, but it is likely to return a 503 Service Unavailable response to the client. In this case, you can still use the `error-page` tag in the `web.xml` file to send the client an appropriate message when the server handles the exception.

Retaining Client and State Information

All but very simple Web applications are likely to require that information of some kind about the client be retained between different page requests. As has been stated, HTTP is stateless and does not provide a mechanism to ascertain that a series of requests have come from the same client.

There are a number of ways of retaining information about clients, such as hidden fields, cookies, and sessions (all described in this section). Sessions are by far the simplest to use, but you should be aware that no one method provides a perfect mechanism, and your servlet may need to be written to support more than one technique.

Using Session Objects

Sessions identify page requests that originate from the same browser during a specified period of time.

Conveniently, sessions are shared by all the servlets in an application accessed by a client.

The `javax.servlet.http.HttpSession` object identifies a session and is obtained using the `HttpServletRequest.getSession()` method. The `HttpSession` object contains the information shown in Table 12.5. This information can be used to identify the session.

TABLE 12.5 Information Accessed Through `HttpSession` Objects

Access Method	*Information*
`getId()`	Unique session identifier
`getLastAccessedTime()`	The last time the client sent a request associated with this session

TABLE 12.5 Continued

Access Method	*Information*
`getCreationTime()`	The time when this session was created
`getMaxInactiveInterval()`	The maximum time interval, in seconds, that the servlet container will keep this session open between client accesses
`getAttribute()`	Objects bound to this session

Behind the scenes, most Web servers implement session tracking using cookies. Information is stored in the cookie to associate a session identifier with a user. The explicit use of cookies to do the same thing is described later in this section.

Creating a Session

To create a session, the servlet must first get a `HttpSession` object for a user. The `HttpServletRequest.getSession()` method returns a user's current session. If there is no current session, it will create one before returning it.

You must call `getSession()` before any output is written to the response (that is, before accessing the `PrintWriter`, not just before sending any HTML or other data to the client).

You can use `HttpSession.isNew()` to determine if a session has been created. The following shows the use of `getSession()` to return a `HttpSession` object:

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
➥throws ServletException, IOException {
    HttpSession session = req.getSession();
    out = res.getWriter();
    if (session.isNew()) {
     /* new session created ok */
... }
}
```

After a session is created, you can start associating objects with the session. The following code could be used in a shopping cart application. It checks to see the user already has a shopping cart in this session. If no cart exists, one is created.

```
HttpSession session = request.getSession();
ShoppingCart cart = (ShoppingCart)session.getAttribute("candystore.cart");
if (cart == null) {
    cart = new ShoppingCart();
    session.setAttribute("candystore.cart", cart);
}
```

Invalidating a Session

A session can either be manually invalidated or allowed to timeout. The default timeout period is 1,800 seconds (30 minutes). A servlet can control the period of time between client requests before the servlet container will invalidate this session with `setMaxInactiveInterval(int seconds)`. Setting a negative time ensures the session will never timeout.

A call to the `invalidate()` method also invalidates a session and unbinds any objects bound to it. This is a useful thing to do if the user logs out of your application.

Hidden Form Fields

Another way of supporting session tracking is to use hidden form fields. These are fields on a HTML page that are not seen by the user. To the server, there is no difference between a hidden field and a non-hidden field. In the browser, hidden fields are not displayed.

The following is an example of a hidden field that could be used to record that the user had read the terms and conditions:

```
<INPUT TYPE=hidden NAME="terms" VALUE="true">
```

Hidden fields have several advantages over other session tracking methods:

- They are supported by all the popular browsers.
- They require no special server support.
- They can be used with anonymous clients.

The major disadvantage is they only work for a sequence of dynamically generated forms, and the technique fails if there is an error before the data is permanently stored somewhere.

The user can modify the values stored in hidden fields, so they are not secure. Do not use hidden fields to hold data that will cause a problem if it is compromised.

Cookies

The `HttpSession` interface and hidden fields provide a simple way to track information during a single session, but they both have the drawback that they cannot be used to retain information across multiple browser sessions. To do this, you must create and manage your own data using cookies.

Any use of cookies comes with a major health warning. As you will see, they are easy to forge and, because they are not always handled consistently, they are inherently unreliable.

You need to be aware that, as a security precaution, a browser can be set up to reject cookies. In this case, you will need to use an additional and alternative method (hidden fields or URL rewriting) to track sessions. Also, cookies should not be used when the information is important or sensitive. Cookies are associated with a browser and can be stored as text in the file system on the client's host. Consequently, cookies are

- Not secure because they are easy to edit or replace
- Potentially can be viewed by other users of the same workstation
- Can allow a user to impersonate another user
- Not available if the user changes his or her workstation or browser

Within a single organization, remember that many users can share machines, such as public access terminals, including part-time or shift workers.

A cookie has a name and a single value, both of which are strings. It may also have optional attributes, such as a comment, path and domain qualifiers, a maximum age, and a version number. You need to be aware that browsers vary as to how they handle cookie attributes, so use them with caution. The browser may also have a limit on the number of cookies it will handle at any one time and may set a limit on its size.

The cookie information is sent as part of the HTTP response header. If the browser accepts cookies, it will store the cookie in the file system on the client.

Creating a Cookie

You can use a `javax.servlet.http.Cookie` object to store information that will remain persistent across multiple HTTP connections.

Cookies can be deleted (sometimes automatically by the browser), so your code should never rely on a cookie being available.

The servlet sends cookies to the browser by using the `HttpServletResponse.addCookie(Cookie)` method.

Because the cookie is sent as part of the HTTP response header, you must add the cookie after setting the response content type but before sending the body of the response. This means a call to `res.addCookie(cookie)` must be made before sending any HTML or other data to the client.

The following code creates a new cookie with a unique identifier (code not shown). In this code fragment, the cookie value is initially left blank and set later using `Cookie.setValue()` to store the URL of the page visited by the user (`getRequestURI()` returns a string containing the URL from the protocol name up to the query string).

```
String userID = new UniqueID();
Cookie cookie = new Cookie (userID, null);
....
String cvalue = getRequestURI();
cookie.setValue(cvalue);
res.addCookie(cookie);
```

By default, a cookie lives only as long as the browser session, so you need to use the `cookie.setMaxAge(interval)` method to change the life expectancy of a cookie. A positive interval sets the number of seconds a cookie will live, which enables you to create cookies that will survive beyond the browser session. A negative interval causes the cookie to be destroyed when the browser exits. An interval of zero immediately deletes the cookie.

Retrieving Cookie Data

Creating a cookie was simple. Retrieving one is not quite as simple. Unfortunately, there is no way to retrieve a cookie by name. Instead, you must retrieve all the cookies and then find the one that interests you. This is yet another good reason for not putting sensitive information in your cookie.

To find the value of a cookie, you use the `Cookie.getValue()` method.

The following code can be used to retrieve a cookie:

```
Cookie cookie = null;
Cookie cookies[] = req.getCookies();
if (cookies != null) {
    int numCookies=cookies.length;
    for(int i = 0; i < numCookies; i++) {
        cookie = cookies[i];
        if (cookie.getName().equals(userID)) {
            String cvalue = cookie.getValue();
            break;
        }
    }
}
```

URL Rewriting

Not all browsers support cookies, and even those that do can be set up to reject cookies. To get around this problem, your servlet can be set up to use URL rewriting as an alternative way of tracking sessions.

With URL rewriting, the server adds extra information dynamically to the URL. Usually, this information is the session ID (a unique ID associated with a `HttpSession` object when it is created).

You can use the `HttpServletResponse.encodeURL()` method to encode the session ID within the response as an added parameter. For example

```
out.println("<FORM action='"+res.encodeURL("/Servlets/HTMLPage")+"'>");
```

adds the session ID to the form's action URL as follows:

```
<FORM action='http://localhost:8000/Servlets/HTMLPage;jsessionid=99B484920067B3'>
```

When the client will not accept a cookie, the server to track sessions will use URL rewriting. To ensure that your servlets support servers that use URL rewriting to track sessions, you must pass all URLs used in your servlet through the `encodeURL()` method.

Servlet Filtering

Filters are a feature that was introduced in version 2.3 of the servlet specification. A filter is a special type of servlet that dynamically intercepts requests and responses and transforms the information contained in them. The main advantage of filters is that they can be added to existing applications without any need for recompilation.

Filters can have several important uses:

- To encapsulate recurring tasks in reusable units
- To format the data sent back to the client
- To provide authorization and blocking of requests
- To provide logging and auditing

A filter can perform filtering tasks on the request, the response, or both.

Programming Filters

A filter is a servlet that also implements the `javax.servlet.Filter` interface.

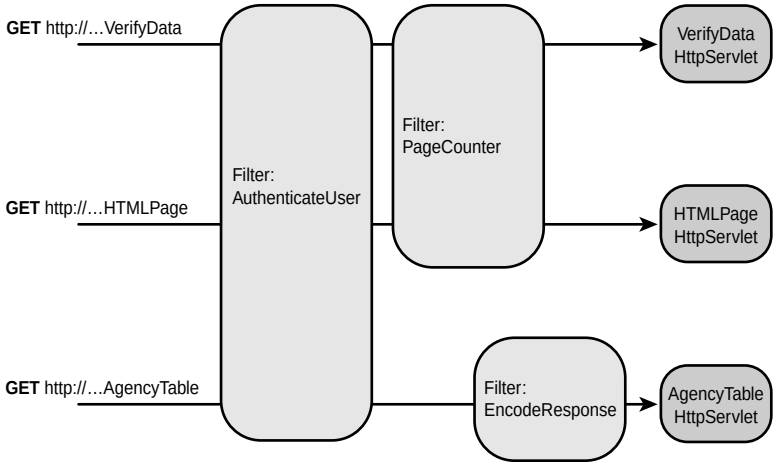
Instead of `doGet()` or `doPost()`, filter tasks are done in a `doFilter()` method.

So that a filter can access initialization parameters, it is passed a `FilterConfig` object in its `init()` method. A filter can access the `ServletContext` through the `FilterConfig` object and thereby load any resources needed for the filtering tasks.

Filters can be connected together in a `FilterChain`. The servlet container constructs the filter chain in the order the filters appear in the deployment descriptor.

Because filters are added to an application at deploy time, it is possible to map a filter to one or more servlets. This is illustrated in Figure 12.17, where the `AuthenticateUser` filter is applied to all servlets. The `PageCounter` is mapped to `VerifyData` and `HTMLPage`, and the `EncodeResponse` filter only affects requests to the `AgencyTable` servlet.

FIGURE 12.17
Servlet filter chain.



After a filter successfully completes its task, it must then call the `doFilter()` method on the next filter in the chain. The filter chain object is passed to the filter by the server as a parameter to the `doFilter()` method.

```
public void doFilter(ServletRequest req, ServletResponse res,
➥FilterChain chain) throws IOException, ServletException {
    ...  // filter code
    chain.doFilter(req, res);  // call the next filter in the chain
}
```

If it is the last filter in the chain, instead of invoking another filter, the container will invoke the resource at the end of the chain (a normal servlet). If for some reason the filter processing fails, and if it is no longer appropriate to continue servicing the request, there is no need to call `doFilter()` on the next filter in the chain.

In a filter, you must provide an `init(FilterConfig)` method.

This `init(FilterConfig)` method is called by the Web server when a filter is being placed into service, and it must complete successfully before the filter can do any filtering work.

The `destroy()` method is called when a filter is being taken out of service, and gives the filter an opportunity to clean up any resources.

The use of these methods will be shown in the following example.

Example Auditing Filter

The filter in Listing 12.9 logs the number of times the application is accessed. A similar technique could be used to intercept requests and authorize the user before calling the servlet to service the request.

LISTING 12.9 Servlet Filter

```
 1: import java.io.*;
 2: import javax.servlet.*;
 3: import javax.servlet.http.*;
 4:
 5: public class AuditFilter extends HttpServlet implements Filter {
 6:     private FilterConfig filterConfig = null;
 7:
 8:     public void init(FilterConfig filterConfig)
 9:        throws ServletException {
10:        this.filterConfig = filterConfig;
11:     }
12:
13:     public void destroy() {
14:        this.filterConfig = null;
15:     }
16:
17:     public void doFilter(ServletRequest req, ServletResponse res,
➥FilterChain chain) throws IOException, ServletException {
18:         if (filterConfig == null)
19:             return;
20:         StringBuffer buf = new StringBuffer();
21:         buf.append ("The number of hits is: ");
22:         synchronized (this) {
23:             Integer counter =
➥(Integer)filterConfig.getServletContext().getAttribute("Counter");
24:             if (counter == null)
25:                 counter = new Integer(1);
26:             else
27:                 counter = new Integer(counter.intValue() + 1);
28:             buf.append (counter.toString());
29:             filterConfig.getServletContext().log(buf.toString());
```

LISTING 12.9 Continued

```
30:             filterConfig.getServletContext().setAttribute("Counter",
➥counter);
31:         }
32:         chain.doFilter(req, res);  // call the next filter in the chain
33:     }
34: }
```

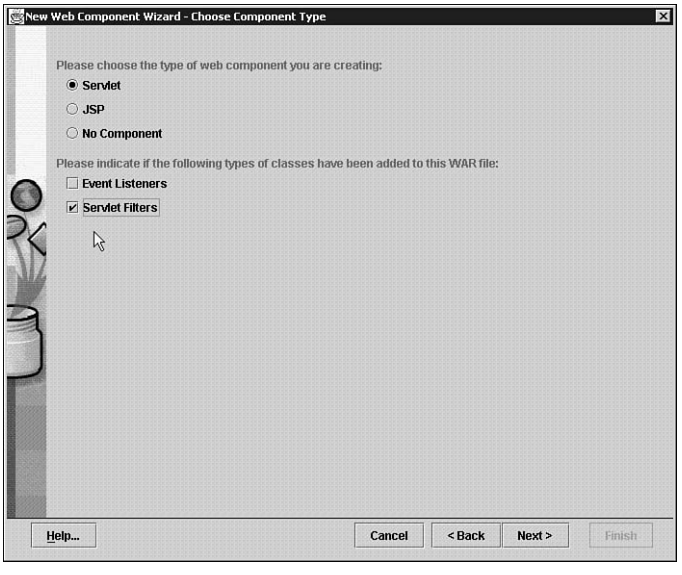
Deploying Filters

Programming the filter is (as always) only half the task. Now it has to be configured into the application. In J2EE RI, this is achieved with the following steps:

1. Add a new Web component to your Servlets application containing the filter class. On the component page, check the Servlets Filters box to indicate that this is a filter class (see Figure 12.18). There is no need to give this servlet a component alias because it is not referenced directly.

FIGURE 12.18

`deploytool` Choose Component Type—Servlet Filters.


New Web Component Wizard - Choose Component Type
Please choose the type of web component you are creating:
Servlet
JSP
No Component
Please indicate if the following types of classes have been added to this WAR file:
Event Listeners
Servlet Filters
Help...
Cancel
< Back
Next >
Finish

2. With the Servlets application highlighted in the left window, select the Filter Mapping tab (see Figure 12.19).
3. Click the Edit Filter List button and add the **`AuditFilter`** to the Servlet Filters box (see Figure 12.20).

FIGURE 12.19

Deploytool Filter Mapping tab.

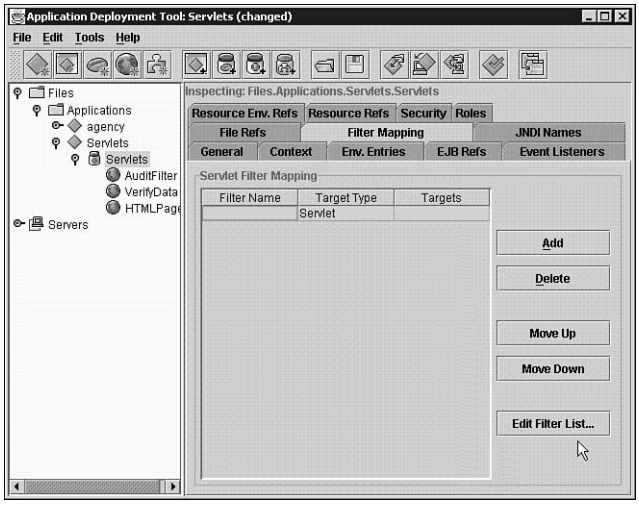
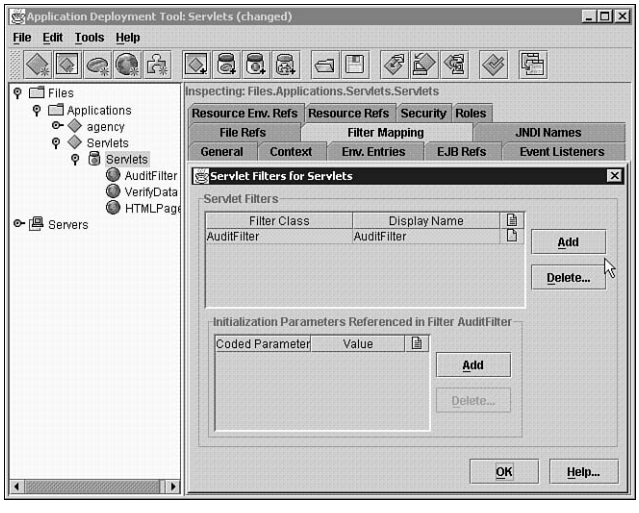


FIGURE 12.20

`deploytool` Servlet filters.



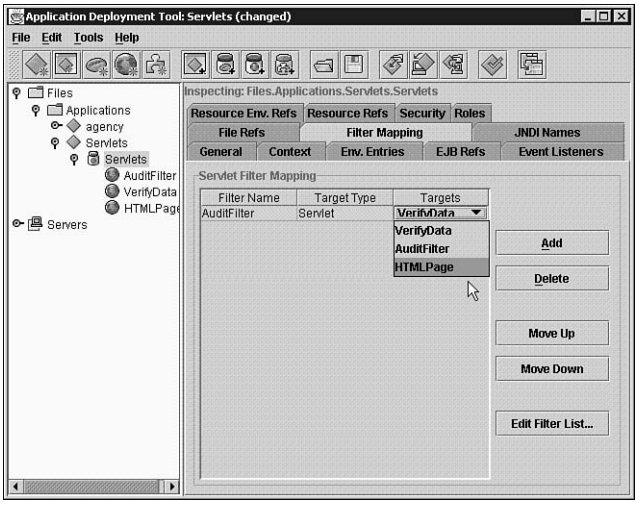
If you view the servlet's deployment descriptor, you will see the following lines have been added:

```
<filter>
    <filter-name>AuditFilter</filter-name>
    <display-name>AuditFilter</display-name>
    <description></description>
    <filter-class>AuditFilter</filter-class>
  </filter>
```

4. Now map the filter to the page whose access you want to be logged. In this case, **`HTMLPage`** (see Figure 12.21).

FIGURE 12.21

`deploytool` Filter Mapping.



The following deployment descriptor entry shows the `AuditFilter` mapped to the `HTMLPage`:

```
<filter-mapping>
  <filter-name>AuditFilter</filter-name>
  <servlet-name>HTMLPage</servlet-name>
</filter-mapping>
```

5. Deploy the application.

Now, every time the `HTMLPage` is accessed, the counter will be incremented and logged. The log file will be found in your J2EE installation directory under `logs\<machine name>\web`.

There will probably be a number of log files in this directory; look for the one with the latest date on it.

At the end of the log file you should see something like the following:

```
HTMLPage: init
The number of hits is: 1
The number of hits is: 2
The number of ....
```

Another way to see if the filter is working is to amend the `HTMLPage` code to output the counter, as shown in Listing 12.10.

LISTING 12.10 `HTMLPage` with Counter Added

```
 1: import java.io.*;
 2: import javax.servlet.*;
 3: import javax.servlet.http.*;
 4:
 5: public class HTMLPage extends HttpServlet {
 6:
 7:     public void doGet(HttpServletRequest req, HttpServletResponse res)
 8:                   throws ServletException, IOException {
 9:         res.setContentType ("image/gif");
10:         PrintWriter out = res.getWriter();
11:         out.println ("<HTML>");
12:         out.println ("<HEAD><TITLE>Filtered Servlet</TITLE></HEAD>");
13:         out.println ("<BODY>");
14:         out.println ("<H1>A Generated HTML Page with a Filter
attached</H1>");
15:         Integer counter =
➥(Integer)getServletContext().getAttribute("Counter");
16:         out.println ("<P>This page has been accessed " +
➥counter + " times</P>");
17:         out.println ("</BODY>");
18:         out.println ("</HTML>");
19:     }
20: }
```

This was a very simple example, more sophisticated filters can be used to modify the request or the response. To do this, you must override the request or response methods by wrapping the request or response in an object that extends `HttpServletRequestWrapper` or `HttpServletResponseWrapper`. This approach follows the well-known Wrapper or Decorator pattern technique. Patters will be described in detail on Day 18, "Patterns."

Event Listening

A servlet can be designated as an event listener. This enables the servlet to be notified when some external event or change has occurred.

There are a number of listener interfaces that you can implement in your servlet. All the listener interfaces extend `java.util.EventListener`.

Table 12.6 provides a list of the listener interfaces.

TABLE 12.6 Servlet Event Listener Interfaces

Listener Interface	*Notification*
`HttpSessionActivationListener`	When a session is activated or passivated
`HttpSessionAttributeListener`	When a session attribute is added, removed, or replaced
`HttpSessionListener`	When a session is created or destroyed
`ServletContextAttributeListener`	When a servlet context attribute is added, removed, or replaced
`ServletContextListener`	About changes to servlet context, such as context initialization or the context is to be shut down

The listener classes are often used as a way of tracking sessions within a Web application. For example, it is often useful to know whether a session became invalid because the Web server timed out the session or because a Web component within the application called the `invalidate()` method.

In the filter section, we incremented a counter each time the application was accessed. The servlet filter code had to check that the counter existed before incrementing it. The following `ServletContextListener` (see Listing 12.11) sets up the counter when the context is initialized before any request is processed. The `ServletContextListener` interface has two methods:

- `contextInitialized()` This receives notification that the application is ready to process requests.
- `contextDestroyed()` This is notified that the context is about to be shut down.

LISTING 12.11 Servlet Listener to Initialize Counter

```
 1: import javax.servlet.*;
 2: import javax.servlet.http.*;
 3:
 4: public class Listener extends HttpServlet
➥implements ServletContextListener {
 5:
 6:    private ServletContext context = null;
 7:
 8:    public void contextInitialized(ServletContextEvent event) {
 9:       context = event.getServletContext();
10:
11:       Integer counter = new Integer(0);
12:       context.setAttribute("Counter", counter);
13:       context.log("Created Counter");
14:    }
15:
```

LISTING 12.11 Continued

```
16:    public void contextDestroyed(ServletContextEvent event) {
17:       event.getServletContext().removeAttribute("Counter");
18:    }
19: }
```

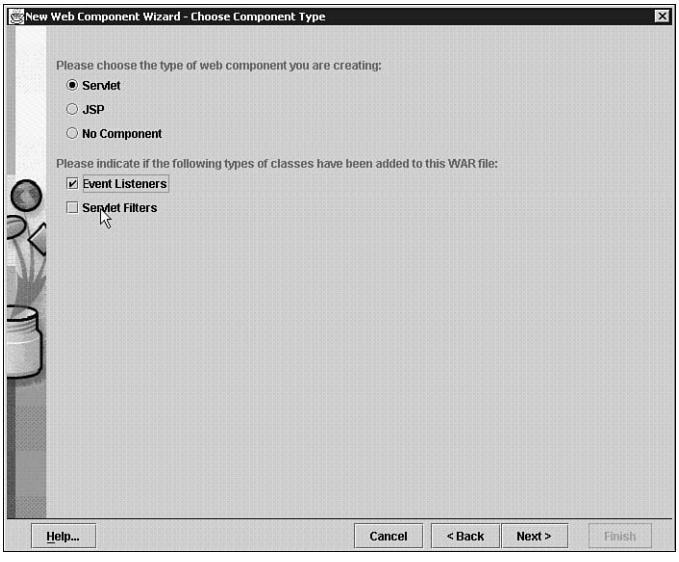
Deploying the Listener

Go through the following steps to deploy the listener class in the Servlets application.

1. Add a new Web component to the Servlets application. On the component page, check the Event Listeners box to indicate that this is an event listener class (see Figure 12.22). As with a filter, there is no need to give the listener servlet a component alias because it is not called directly.

FIGURE 12.22

`deploytool` Choose Component Type—Event Listeners.


New Web Component Wizard - Choose Component Type
Please choose the type of web component you are creating:
Servlet
JSP
No Component
Please indicate if the following types of classes have been added to this WAR file:
Event Listeners
Servlet Filters
Help...
Cancel
< Back
Next >
Finish

2. With the Servlets application highlighted in the left window, select the Event Listeners tab (see Figure 12.23).
3. Add the Listener class to the event listeners (see Figure 12.24).

   The following will then be added to the Servlets deployment descriptor:

```
<listener>
    <listener-class>Listener</listener-class>
  </listener>
```

FIGURE 12.23

`deploytool` Event Listeners.

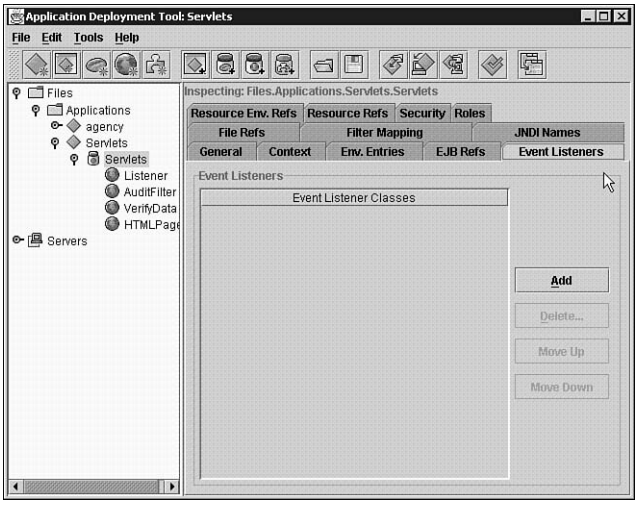
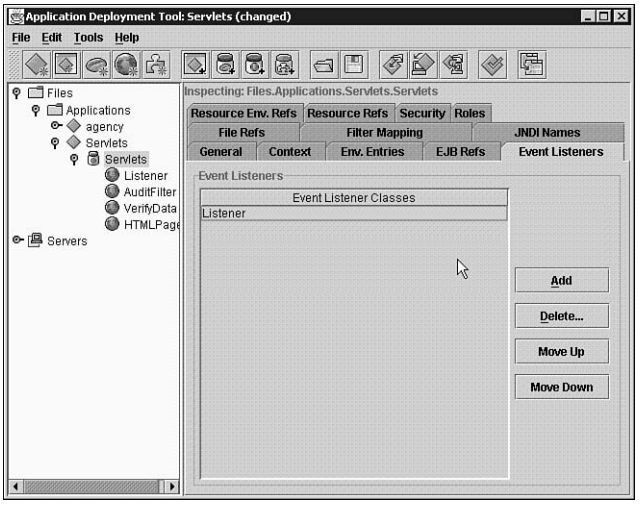


FIGURE 12.24

`deploytool` Event Listeners.



4. Deploy the application.

Because the listener initializes the counter to 0, and this is guaranteed to be called before any servlet code, you can now simplify the `AuditFilter` code and remove the check for a `null` counter (see Listing 12.12).

LISTING 12.12 Amended `AuditFilter` Code Showing New `doFilter()` Method

```
   1: public void doFilter(ServletRequest req, ServletResponse res,
➥FilterChain chain) throws IOException, ServletException {
   2:     if (filterConfig == null)
   3:         return;
   4:     StringBuffer buf = new StringBuffer();
   5:     buf.append ("The number of hits is: ");
   6:     synchronized (this) {
   7:         Integer counter =
➥(Integer)filterConfig.getServletContext().getAttribute("Counter");
   8:         counter = new Integer(counter.intValue() + 1);
   9:         buf.append (counter.toString());
  10:         filterConfig.getServletContext().log(buf.toString());
  11:         filterConfig.getServletContext().setAttribute("Counter", counter);
  12:     }
  13:     chain.doFilter(req, res);
  14: }
```

Servlet Threads

In the `AuditFilter` example, several lines of code were encased in a synchronize block. This was done to ensure that the counter would be correctly incremented if more than one access is made to the page at the same time. Without the synchronize block, it is possible for two (or possibly more) servlet threads to obtain the counter value before it has been incremented, thereby losing one of the increments. In this case, failing to count one page hit is not a significant problem. There are other situations where it might be.

Shared resources, such as files and databases, can also present concurrency issues when accessed by more than one servlet at a time. To avoid concurrency issues, a servlet can implement *the single-thread model*.

The servlet `init()` method is only called once, so concurrency is not a problem here.

The servlet API specifies that a servlet container must guarantee that no two threads of a servlet that implements the `javax.servlet.SingleThreadModel` interface are run concurrently. This means the container must maintain a pool of servlet instances and dispatch each new request to a free servlet; otherwise, only one request at a time can be handled.

If you want to make your servlet single threaded to prevent concurrency problems, simply add the `SingleThreadModel` interface to the class signature.

```
public class HTMLPage extends HttpServlet implements SingleThreadModel {
```

Security and the Servlet Sandbox

A servlet runs within the Web server and, if allowed, can access the file system and network or could even call `System.exit()` to shutdown the Web server. Giving a servlet this level of trust is not advisable, and most Web servers run servlets in a sandbox, which restricts the damage a rogue servlet could potentially cause.

A servlet sandbox is an area where servlets are given restricted access to the server. Servlets running in the sandbox can be constrained from accessing the file system and network. This is similar to how Web browsers control applets. The implementation of the sandbox is server dependent, but a servlet in a sandbox is unlikely to be able to

- Access server files
- Access the network
- Run commands on the server

Agency Case Study

You will now add a servlet to the Agency case study. The servlet sends the contents of the agency database tables to be displayed by a browser. The rows of the tables are formatted as an HTML table.

The servlet first outputs an HTML form on which the user selects the name of a table from a pull-down list (see Figure 12.25).

After the user clicks the Submit Query button, the servlet displays the contents of this table on the same page.

`AgencyTable` Servlet Code

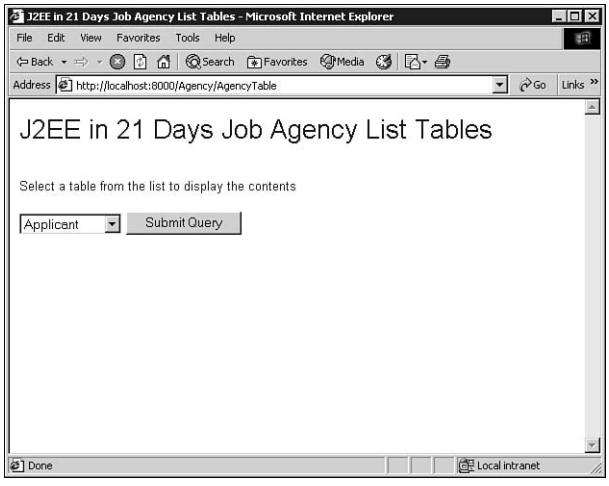
The `AgencyTable` servlet overrides the `init()` method to obtain the servlet context along with the JNDI context for the Agency EJB. The Agency EJB `select()` method is used in the `doGet()` method to obtain the rows of table data from the `Agency` database.

```
public void init(){
    context = getServletContext();
    try {
        InitialContext ic = new InitialContext();
        Object lookup = ic.lookup("java:comp/env/ejb/Agency");
```

```
        AgencyHome home = (AgencyHome)PortableRemoteObject.narrow(lookup,
➥AgencyHome.class);
        agency = home.create();
    }
....
}
```

FIGURE 12.25

AgencyTable form.


J2EE in 21 Days Job Agency List Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8000/Agency/AgencyTable
Go Links
J2EE in 21 Days Job Agency List Tables
Select a table from the list to display the contents
Applicant
Submit Query
Done
Local intranet

The `AgencyTable` HTML page is generated in the following code.

```
private final String tables = "<OPTION>Applicant<OPTION>ApplicantSkill
➥<OPTION>Customer<OPTION>Job<OPTION>JobSkill
➥<OPTION>Location<OPTION>Matched<OPTION>Skill";
...

out.println ("<HTML>");
out.println ("<HEAD><TITLE>" + agencyName + " List Tables</TITLE></HEAD>");
out.println ("<BODY><FONT FACE=ARIAL COLOR=DARKBLUE");
out.println ("<H1><FONT SIZE=+3>" + agencyName + " List Tables</FONT></H1>");
out.println ("<P>Select a table from the list to display the contents</P>");

out.println ("<FORM>");
out.println ("<SELECT NAME=\"tableList\" SIZE=1>" + tables + "</SELECT>");
out.println ("<INPUT TYPE=submit>");
out.println ("</FORM>");
out.println ("</FONT></BODY>");
out.println ("</HTML>");
```

The String variable `tables` contains the names of the database tables encoded in a HTML `<OPTION>` list.

The <SELECT> tag defines a parameter called tableList. This parameter is used to pass the name of the selected table from the form to the servlet.

The following code checks to see if the parameter has been set. If it has, the outputTable() method is called.

```
tableName = req.getParameter("tableList");
if (tableName != null) {
    outputTable(out, tableName, res);
```

In the outputTable() method, it is the agency.select bean that actually does the work. It returns a list of all the rows in the table.

```
java.util.List query;   // list of String[], first row = column names
query = agency.select(tableName);
```

The rows of the table are displayed as an HTML table so the columns line up correctly. The attributes to the HTML TABLE tag set the border and background colors, and the cell padding is increased to improve readability. Rows in a table are separated by <TR>...</TR> tags.

```
out.println ("<TABLE BORDER=1 BORDERCOLOR=SILVER BGCOLOR=IVORY
➥CELLPADDING=5><TR>");
```

The first item in the list is an array containing the names of the columns in the table. This is output as HTML table header cells.

```
String[] headerRow = (String[])query.get(0);
for (int i = 0; i < headerRow.length; i++) {
    out.println ("<TH ALIGN=LEFT>" + headerRow[i] + "</TH>");
}
out.println ("</TR>");
```

The remainder of the rows of the table are output as HTML table data cells.

```
for (int i = 1; i < query.size(); i++) {
    out.println ("<TR>");
    String[] row = (String[])query.get(i);
    for (int r = 0; r < row.length; r++) {
        out.println ("<TD>" + row[r] + "</TD>");
    }
    out.println ("</TR>");
}
out.println ("</TABLE>");
```

Deploying the AgencyTable Servlet

Use deploytool to add the AgencyTable Servlet as a servlet Web component in the Agency application. You will need to map the JNDI names used in the code, as shown in Figure 12.26.

FIGURE 12.26

`deploytool` EJB Refs.

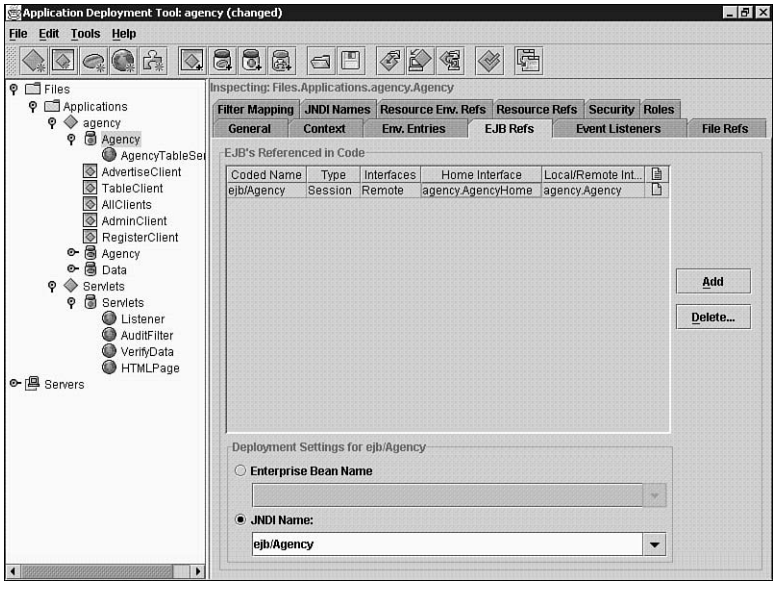
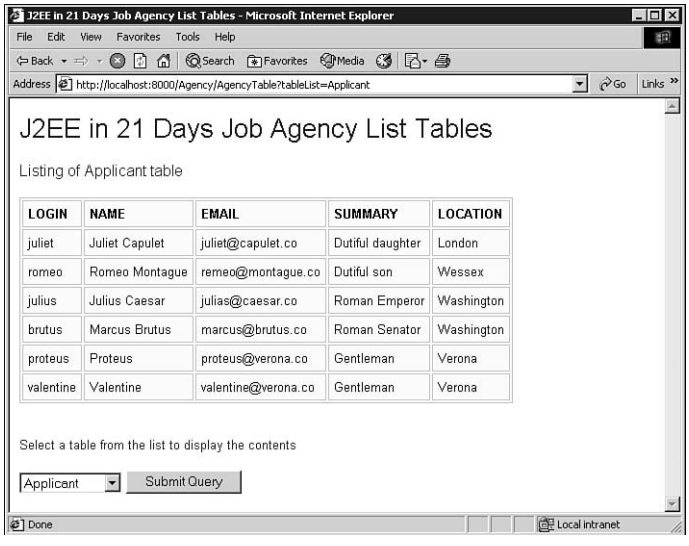


Figure 12.27 shows the output.

FIGURE 12.27

`AgencyTable` servlet output.


J2EE in 21 Days Job Agency List Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8000/Agency/AgencyTable?tableList=Applicant
Go Links
J2EE in 21 Days Job Agency List Tables
Listing of Applicant table
LOGIN NAME EMAIL SUMMARY LOCATION
juliet Juliet Capulet juliet@capulet.co Dutiful daughter London
romeo Romeo Montague remeo@montague.co Dutiful son Wessex
julius Julius Caesar julias@caesar.co Roman Emperor Washington
brutus Marcus Brutus marcus@brutus.co Roman Senator Washington
proteus Proteus proteus@verona.co Gentleman Verona
valentine Valentine valentine@verona.co Gentleman Verona
Select a table from the list to display the contents
Applicant
Submit Query
Done
Local intranet

The complete listing of the AgencyTable servlet is provided in Listing 12.13 for completeness.

LISTING 12.13 AgencyTable Servlet Code

```
 1: import java.io.*;
 2: import javax.servlet.*;
 3: import javax.servlet.http.*;
 4: import agency.*;
 5: import javax.naming.*;
 6: import java.rmi.*;
 7: import javax.rmi.*;
 8: import javax.ejb.*;
 9:
10: public class AgencyTableServlet extends HttpServlet
11: {
12:     private final String tables = "<OPTION>Applicant<OPTION>ApplicantSkill
➥<OPTION>Customer<OPTION>Job<OPTION>JobSkill
➥<OPTION>Location<OPTION>Matched<OPTION>Skill";
13:     private Agency agency;
14:     private ServletContext context;
15:
16:     public void init(){
17:         context = getServletContext();
18:         try {
19:             InitialContext ic = new InitialContext();
20:             Object lookup = ic.lookup("java:comp/env/ejb/Agency");
21:             AgencyHome home =
➥(AgencyHome)PortableRemoteObject.narrow(lookup, AgencyHome.class);
22:             agency = home.create();
23:         }
24:         catch (NamingException ex) {
25:             context.log("NamingException in AgencyTableServlet.init", ex);
26:         }
27:         catch (ClassCastException ex) {
28:             context.log("ClassCastException in AgencyTableServlet.init",
➥ex);
29:         }
30:         catch (CreateException ex) {
31:             context.log("CreateException in AgencyTableServlet.init", ex);
32:         }
33:         catch (RemoteException ex) {
34:             context.log("RemoteException in AgencyTableServlet.init", ex);
35:         }
36:     }
37:
38:     public void destroy () {
39:         context = null;
40:         agency = null;
```

LISTING 12.13 Continued

```
    }

    private void outputTable (PrintWriter out, String tableName,
➥HttpServletResponse res) throws RemoteException{

        java.util.List query;   // first row = column names
        query = agency.select(tableName);

        out.println ("<P><FONT SIZE=+1>Listing of " + tableName +
➥" table</FONT></P>");
        out.println ("<TABLE BORDER=1 BORDERCOLOR=SILVER BGCOLOR=IVORY
➥CELLPADDING=5><TR>");

        String[] headerRow = (String[])query.get(0);
        for (int i = 0; i < headerRow.length; i++) {
            out.println ("<TH ALIGN=LEFT>" + headerRow[i] + "</TH>");
        }
        out.println ("</TR>");

        for (int i = 1; i < query.size(); i++) {
            out.println ("<TR>");
            String[] row = (String[])query.get(i);
            for (int r = 0; r < row.length; r++) {
                out.println ("<TD>" + row[r] + "</TD>");
            }
            out.println ("</TR>");
        }
        out.println ("</TABLE>");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
                 throws IOException {
        try {
            String agencyName = agency.getAgencyName();
            String tableName = null;

            res.setContentType ("text/html");
            PrintWriter out = res.getWriter();

            // print out form
            out.println ("<HTML>");
            out.println ("<HEAD><TITLE>" + agencyName +
➥" List Tables</TITLE></HEAD>");
            out.println ("<BODY><FONT FACE=ARIAL COLOR=DARKBLUE");
            out.println ("<H1><FONT SIZE=+3>" + agencyName +
➥" List Tables</FONT></H1>");

            tableName = req.getParameter("tableList");
            if (tableName != null) {
```

LISTING 12.13 Continued

```
  85:                outputTable(out, tableName, res);
  86:             }
  87:             out.println ("<P><BR>Select a table from the list
➥to display the contents</BR></P>");
  88:             out.println ("<FORM>");
  89:             out.println ("<SELECT NAME=\"tableList\" SIZE=1>" +
➥tables + "</SELECT>");
  90:             out.println ("<INPUT TYPE=submit>");
  91:             out.println ("</FORM>");
  92:
  93:             out.println ("</FONT></BODY>");
  94:             out.println ("</HTML>");
  95:         }
  96:         catch (RemoteException ex) {
  97:             context.log ("RemoteException in AgencyTableServlet.doGet",
➥ex);
  98:             res.sendError (res.SC_INTERNAL_SERVER_ERROR);
  99:         }
 100:     }
 101: }
```

Use a browser to access your servlet and check that you can retrieve the data from the database.

Summary

Today, you have seen how servlets can be employed in a Web application to add dynamic content to HTML pages. You learned that servlets have no client interface, and the servlet container controls its lifecycle. Because HTTP is a stateless protocol, servlets have to use external means to retain information between page accesses. Cookies are one method, but when cookies cannot be used, a servlet can use hidden fields or URL rewriting. You have also seen that with event listening and using servlet filters, you can further extend the functionality and reusability of your servlet Web applications.

Servlets generate HTML from within Java code. This works well when the amount of HTML is relatively small, but the coding can become onerous if large amounts of HTML have to be produced. Tomorrow, you will look at another type of servlet called a Java Server Page (JSP). With JSPs, the opposite approach is taken. Here, the servlet Java code is imbedded in the HTML page, avoiding the need to have multiple `out.println()` statements.

Q&A

Q What are the two main HTTP methods used to send requests to a Web server? What is the main difference between them? Which should I use to send sensitive information to the server?

A The two main methods are `GET` and `POST`. `GET` adds any request parameters to the URL query string, whereas `POST` sends its parameters as part of the request body. It is for this reason that you should use `POST` to send sensitive information.

Q What are the main uses for a `ServletContext` object?

A The main uses are to set and store attributes, log events, obtain URL references to resources, and get the MIME type of files.

Q What are the names of the methods I must implement to handle HTTP `GET` and `POST` requests?

A The methods are `doGet()` and `doPost()`.

Q What are the main uses of a servlet filter?

A Filters can be used to provide auditing and to change the format of the request or the response.

Exercises

To extend your knowledge of Servlets, try the following exercise.

1. Extend the Agency case study. Add a servlet that produces a page that can be used to add new customers to the database. The information you will need to collect is the applicant's name, login, and e-mail address. Use the `agency.createCustomer()` method to store this information in the `Customer` table in the database. Check that your servlet works correctly by running your `AgencyTableServlet` and to display the contents of the `Customer` table.
2. Now add a new servlet to delete a customer. The only information you will need to obtain is the login name. Use the `agency.deleteCustomer` method. Again, check that your servlet works correctly by running your `AgencyTableServlet`.

WEEK 2

DAY 13

JavaServer Pages

Yesterday, you looked at developing Web applications using Java servlets. Servlets have the advantage of being able to generate the HTML Web page dynamically. The disadvantage of servlets is the fact that the developer must generate a lot of HTML formatting information from within Java. Servlets can be described as large amounts of boring HTML `println()` statements interspersed with small amounts of interesting Java code.

Servlets make it difficult to differentiate the presentation layer from the logic layer of an application. This duality of purpose means that servlets do not allow the roles of HTML designer and Java programmer to be easily separated. Writing servlets requires the members of the development team to be either

- Java programmers who must learn HTML and Web design
- Web designers who must learn Java

In practice, there are very few Java programmers who make good Web designers, and fewer Web designers who make good Java programmers.

JavaServer Pages are servlets that are written in HTML. Actually, there is a bit more to it than that, but the Java code on a JSP is usually either non-existent or very simple, and can be readily understood by non-Java programmers.

In today's lesson, you will learn

- What a JSP is and how its implementation differs from servlets
- The JSP lifecycle—what you have to do and what the Web server will do for you
- How to deploy a JSP
- How to use JavaBeans to hide Java functionality from the JSP
- How to develop a Web application using JSPs

Today's work builds directly on the knowledge gained yesterday because many of the mechanisms used in JSPs are the same as servlets.

What is a JSP?

A JSP is just another servlet, and like HTTP servlets, a JSP is a server-side Web component that can be used to generate dynamic Web pages.

The fundamental difference between servlets and JSPs is

- Servlets generate HTML from Java code.
- JSPs embed Java code in static HTML.

To illustrate this difference, Listings 13.1 and 13.2 are the same Web page coded as a servlet and as a JSP, respectively. Each Web page simply reads a parameter called `name` from the HTTP request and creates an HTML page displaying the value of the name parameter. Listing 13.1 shows the servlet, and Listing 13.2 shows the JSP.

LISTING 13.1 Simple Dynamic Page As a Servlet

```
 1: import java.io.*;
 2: import javax.servlet.*;
 3: import javax.servlet.http.*;
 4:
 5: public class Hello extends HttpServlet {
 6:
 7:     public void doGet(HttpServletRequest req, HttpServletResponse res)
 8:                   throws ServletException, IOException {
 9:         res.setContentType ("text/html");
10:         PrintWriter out = res.getWriter();
11:         String name = req.getParameter("name");
12:         out.println ("<HTML>");
13:         out.println ("<HEAD><TITLE>Hello</TITLE></HEAD>");
14:         out.println ("<BODY>");
15:         out.println ("<H1>Hello " + name + "</H1>");
16:         out.println ("</BODY>");
17:         out.println ("</HTML>");
18:     }
19: }
```

LISTING 13.2 Same Dynamic Page As a JSP

```
1: <HTML>
2:   <HEAD><TITLE>Hello</TITLE></HEAD>
3:   <BODY>
4:     <% String name = request.getParameter("name"); %>
5:     <H1>Hello <%=name%> </H1>
6:   </BODY>
7: </HTML>
```

Here you can see that the JSP not only requires far less typing, but a lot of the work is being done for you. How the JSP achieves the same effect with far less code will soon become clear.

Separating Roles

With a servlet, the Java programmer was forced to generate the entire HTML. With a JSP, it is much easier to separate the HTML from the application tasks. With the use of JavaBeans and JSP tag libraries (covered on Day 14, "JSP Tag Libraries"), this separation is even more explicit.

Using JSPs, the Web designer can concentrate on the design and development of the HTML page. When a dynamic element is needed, the developer can use a pre-written bean or tag library to provide the data. The Java programmer can concentrate on developing a useful set of beans and tag libraries encapsulating the complex Java required to retrieve the data from an EJB, a database, or any other data source.

Translation and Execution

JSPs differ from servlets in one other respect. Before execution, a JSP must be converted into a Java servlet. This done in two stages:

1. The JSP text is translated into Java code.
2. The Java code is compiled into a servlet.

The resulting servlet processes HTTP requests. The translate and compile process is performed once before the first HTTP request can be processed. The JSP lifecycle is covered in more detail later.

JSP Syntax and Structure

Before writing your first JSP, you need to gain an understanding of the syntax and the structure of a JSP.

As you have seen, JSP elements are embedded in static HTML. Like HTML, all JSP elements are enclosed in open and close angle brackets (< >). Unlike HTML, but like XML, all JSP elements are case sensitive.

JSP elements are distinguished from HTML tags by beginning with either <% or <jsp:. JSPs follow XML syntax, they all have a start tag (which includes the element name) and a matching end tag. Like XML tags, a JSP tag with an empty body can combine the start and end tags into a single tag. The following is an empty body tag:

```
<jsp:useBean id="agency" class="web.AgencyBean">
</jsp:useBean>
```

The following tag is equivalent to the previous example:

```
<jsp:useBean id="agency" class="web.AgencyBean"/>
```

Optionally, a JSP element may have attributes and a body. You will see examples of all these types during today's lesson. See Appendix C, "An Overview of XML," for more information on XML and the syntax of XML elements.

JSP Elements

The basic JSP elements are summarised in Table 13.1.

TABLE 13.1 JSP Elements

Element Type	*JSP Syntax*	*Description*
Directives	<%@Directive…%>	Information used to control the translation of the JSP text into Java code
Scripting	<% %>	Embedded Java code
Actions	<jsp: >	JSP-specific tags primarily used to support JavaBeans

You will learn about scripting elements first and then cover directives and actions later in today's lesson.

Scripting Elements

Scripting elements contain the code logic. It is these elements that get translated into a Java class and compiled. There are three types of scripting elements—declarations, scriptlets, and expressions. They all start with <% and end with %>.

Declarations

Declarations are used to introduce one or more variable or method declarations, each one separated by semicolons. A variable must be declared before it is used on a JSP page. Declarations are differentiated from other scripting elements with a `<%!` start tag. An example declaration that defines two variables is as follows:

```
<%! String color = "blue"; int i = 42; %>
```

You can have as many declarations as you need. Variables and methods defined in declarations are declared as instance variables outside of any methods in the class.

Expressions

JSP expressions are single statements that are evaluated, and the result is cast into a string and placed on the HTML page. An expression is introduced with <%= and must not be terminated with a semi-colon. The following is an expression that will put the contents of the `i` element in the `items` array on the output page.

```
<%= items[i] %>
```

JSP expressions can be used as values for attributes in JSP tags. The following example shows how the `i` element in the `items` array can be used as the value for a submit button on a form:

```
<INPUT type=submit value="<%= items[i] %>">
```

There cannot be a space between the <% and = signs, because this would cause a syntax error in the JSP (see the later section on the "JSP Lifecycle" for how JSP errors are detected and reported).

Scriptlets

Scriptlets contain code fragments that are processed when a request is received by the JSP. Scriptlets are processed in the order they appear in the JSP. They need not produce output.

Scriptlets can be used to create local variables, for example

```
<% int i = 42;%>
<BIG>The answer is <%= i %></BIG>
```

The difference between scriptlet variables and declarations is that scriptlet variables are scoped for each request. Variables created in declarations can retain their values between requests (they are instance variables).

JSP Comments

There are three types of comments in a JSP page. The first type is called a JSP comment. JSP comments are used to document the JSP page. A JSP comment is completely ignored; it is not included in the generated code. A JSP comment looks like the following:

```
<%-- this is a JSP comment --%>
```

An alternative way to comment a JSP is to use the comment mechanism of the scripting language, as in the following:

```
<% /* this is a java comment */ %>
```

This comment will be placed in the generated Java code.

The third mechanism for adding comments to a JSP is to use HTML comments.

```
<!-- this is an HTML comment -->
```

HTML comments are passed through to the client as part of the response. As a result, this form can be used to document the generated HTML document. Dynamic information can be included in HTML comments as shown in the following:

```
<!-- comment  <%= expression %> comment  -->
```

First JSP example

You are now ready to write and deploy your first JSP. The JSP in Listing 13.3 is a very simple JSP that uses an expression to generate the current date.

LISTING 13.3 Full Text of `date.jsp`

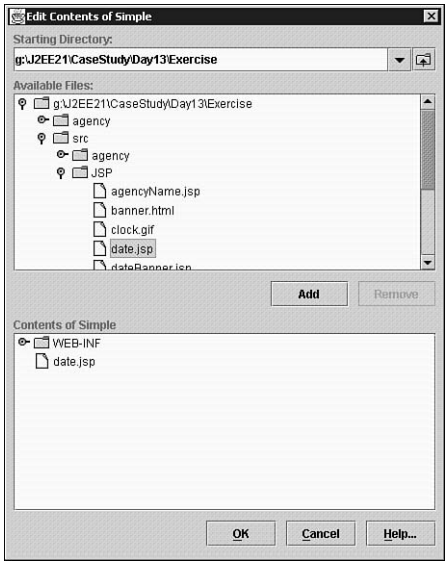
```
 1: <HTML>
 2:   <HEAD>
 3:     <TITLE>JSP Date Example</TITLE>
 4:   </HEAD>
 5:   <BODY>
 6:     <BIG>
 7:       Today's date is <%= new java.util.Date() %>
 8:     </BIG>
 9:   </BODY>
10: </HTML>
```

Perform the following steps to deploy this JSP.

1. Start up the J2EE RI and run `deploytool`.
2. Create a new application to store your JSPs. Call it **`simple`**.
3. Select File, New, Web Component and create a new `WAR` file in the application, call it **`Simple`**.
4. Click Edit and add the JSP file **`date.jsp`**. It does not matter if your JSP source file is in a sub-directory or not, it will be placed at the top-level directory of the Web application. Figure 13.1 shows how the `date.jsp` file in a sub-directory of `src/JSP` is added to the Web application.

FIGURE 13.1

Adding a JSP to a Web component.



5. Click Next.
6. On the Choose Component Type page, set the Web component to be JSP.
7. Click Next and in the JSP Filename box, select `date.jsp`. Leave the component name and display names with default values. Your screen should look like the one shown in Figure 13.2.
8. Click Next twice to get to the Aliases screen.
9. On the Aliases page, add the alias **`/date`**. This alias is used in the URL to find the JSP in the same way as for servlets. Your screen should look like the one shown in Figure 13.3.

FIGURE 13.2

`deploytool` Choose JSP General Properties screen.

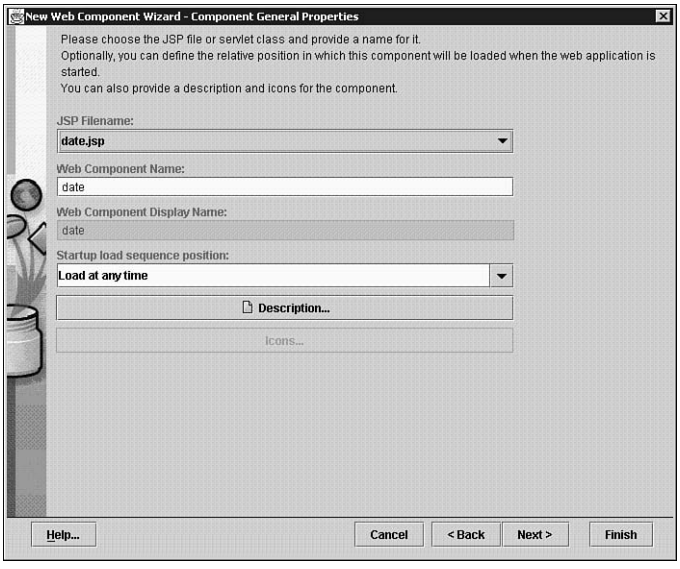
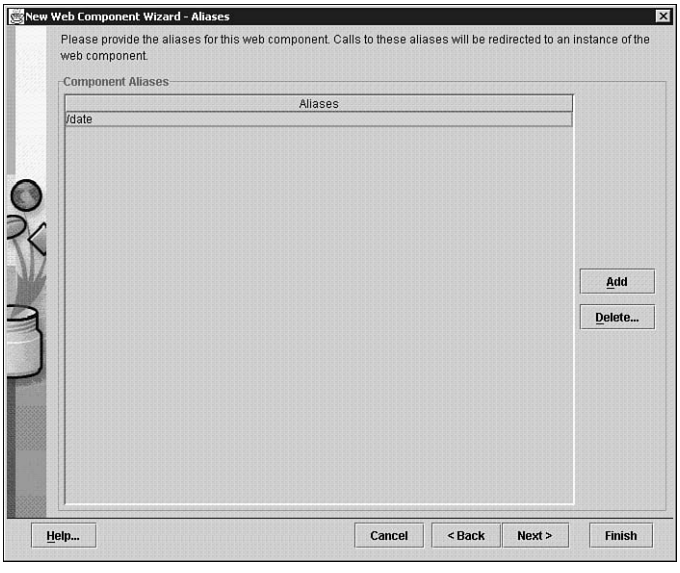

New Web Component Wizard - Component General Properties
Please choose the JSP file or servlet class and provide a name for it.
Optionally, you can define the relative position in which this component will be loaded when the web application is started.
You can also provide a description and icons for the component.
JSP Filename:
date.jsp
Web Component Name:
date
Web Component Display Name:
date
Startup load sequence position:
Load at any time
Description...
Icons...
Help...
Cancel
< Back
Next >
Finish

FIGURE 13.3

`deploytool` JSP Alias page.



10. Click Finish.
11. Select Tools, Verifier to check the application before deployment. Ignore any warning about a missing context because you will add this in the next step.

12. Select Tools, Deploy to deploy the application. Step through each deployment screen and supply a context root of `/simple` on the War Context Root screen.

As with servlets, the URL to reference the JSP under the J2EE RI is as follows:

```
http://<Web server address>/<Context root>/<Component alias>
```

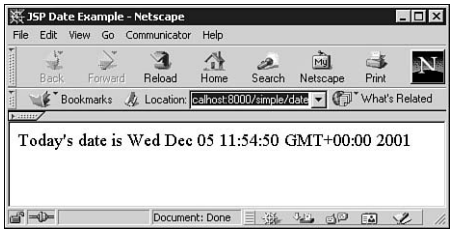
For this example, the URL must specify the local host with a port number of 8000 because the J2EE Web server does not use the standard HTTP port number. A suitable URL follows:

```
http://localhost:8000/simple/date
```

Enter this URL in your favorite browser. As long as you did not make an error copying Listing 13.3, you should see the page shown in Figure 13.4. Each time you refresh the page, the time should change by a few seconds.

FIGURE 13.4

Browser showing the date JSP.


JSP Date Example - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: calhost:8000/simple/date What's Related
Today's date is Wed Dec 05 11:54:50 GMT+00:00 2001
Document: Done

JSP Problems

There are three types of errors you can make with JSP pages:

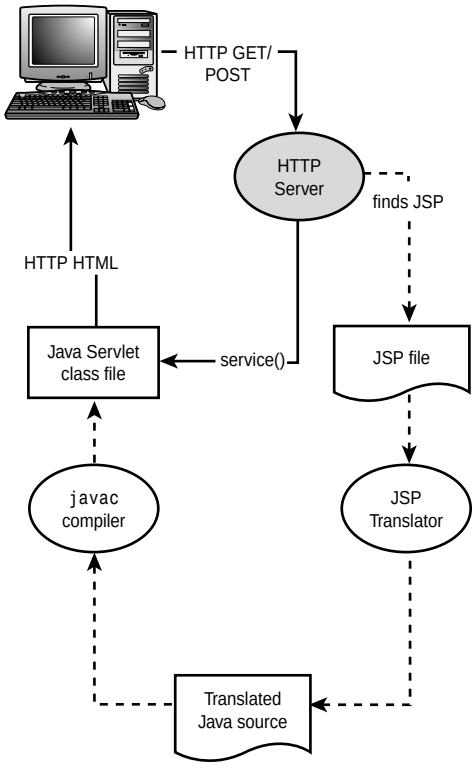
- JSP errors causing the translation to fail
- Java errors causing the compilation to fail
- HTML errors causing the page to display incorrectly

Finding and correcting these errors can be quite problematic because the information you need to discover the error is not readily available. Before looking at resolving errors, you will need to understand the JSP lifecycle.

JSP Lifecycle

As has already been stated, JSPs go through a translation and compilation phase prior to processing their first request. This is illustrated in Figure 13.5.

FIGURE 13.5
JSP translation and processing phase.



The Web server automatically translates and compiles a JSP; you do not have to manually run any utility to do this. JSP translation and compilation can occur at any time prior to the JSP first being accessed. It is implementation dependent when this translation and compilation occurs but it is usually either

- On deployment
- When the first request for the JSP is received

If the latter strategy is used, not only is there a delay in processing the first request because the page is translated and compiled, but if the compilation fails, the client will be presented with some unintelligible error. If your server uses this strategy, ensure that you always force the translation and compilation of your JSP, either by making the first page request after it has been deployed or by forcing the page to be pre-compiled.

With J2EE RI, the translation and compilation only takes place when the page is first accessed. You can find the translated JSP in `<J2EE installation>\repository\<machine name>\web\<context root>\`. You may find it useful to refer to the translated JSP to understand any compilation errors.

With J2EE RI, you can force the page to be pre-compiled by using your Web browser and appending `?jsp_precompile=true` to the JSP's URL string. To pre-compile the `date.jsp` example, you could use the following:

```
http://localhost:8000/simple/date?jsp_precompile=true
```

Because the compiled servlet is not executed, there are several advantages:

- There is no need to add any page-specific parameters to the URL.
- Pages do not have to be compiled in an order determined by the application logic.
- In a large application, less time is wasted traversing already compiled pages to find non-compiled ones, and it is easier to ensure that pages are not missed.

Detecting and Correcting JSP Errors

Realistically, you are going to make errors when writing JSPs. These errors can be quite difficult to comprehend because of the way they are detected and reported. There are three categories of error:

- JSP translation
- Servlet compilation
- HTML presentation

The first two categories of error are detected by the Web server and sent back to the client browser instead of the requested page. The last type of error (HTML) is detected by the Web browser.

Correcting each category of error requires a different technique and is discussed in this section.

Translation Errors

If you mistype the JSP tags or fail to use the correct attributes for the tags, you will get a translation error returned to your browser. With the simple date example, missing the closing `%` sign from the JSP expression, as in the following code

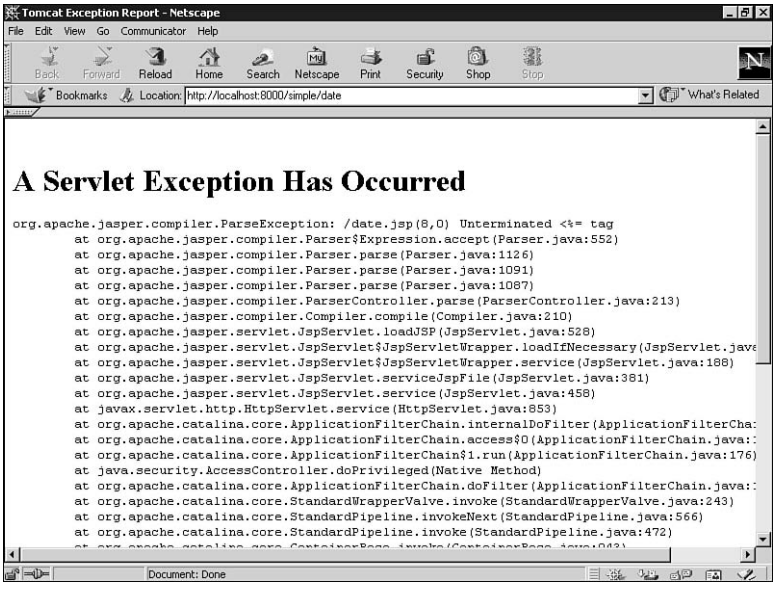
```
Today's date is <%= new java.util.Date() >
```

will generate a translation error. Figure 13.6 shows the date JSP page with a translation error.

Using the Web browser to report errors is an expedient solution to the problem of reporting errors, but this approach is not used by all Web servers. Some simply write the error to a log file and return an HTTP error to the browser. The JSP specification simply requires the Web server to report an HTTP 500 problem if there is an error on the JSP.

FIGURE 13.6

Browser showing JSP translation error.



The error in Figure 13.6 shows a parse error defining an unterminated "`<%=` tag" on line 8 of the "`date.jsp`" page. The first line of the error is

```
org.apache.jasper.compiler.ParseException: /date.jsp(8,0) Unterminated <%= tag
```

This shows all of the useful information for determining the error. The first part of the line tells you the exception that occurred:

```
org.apache.jasper.compiler.ParseException:
```

In this case, a generic parsing exception reported by the JSP translator. The J2EE RI includes a version of the Apache Tomcat Web server and it is the Jasper parser of Tomcat that has reported the error.

The second part of the error identifies the JSP page:

```
/date.jsp
```

and the third part specifies the line and column number:

```
(8,0)
```

You know that the error is on line 8 of the `date.jsp` page. The column number is often misleading and is best ignored when looking for the error.

The final part of the error message is a brief description of the problem:

```
Unterminated <%= tag
```

The rest of the error information returned to the Web browser is a stack trace of where the exception occurred in the Jasper translator. This is of no practical use to you and can be ignored.

From the error information you should be able to identify the problem on the original JSP. Depending on the nature of the error, you may need to look at JSP lines prior to the one with the reported error. Sometimes errors are not reported until much later in the JSP. The worst scenario is when the error is reported on the very last line because this means the error could be practically anywhere in the JSP.

Compilation Errors

Compilation errors can occur when you mistype the Java code in a Java scripting element or when you omit necessary page directives, such as import package lists (see the next section).

Compilation errors are shown on the page returned to the browser and show the line number in error in the generated file. Figure 13.7 shows compilation error that occurs if you mistype `Date` as `Datex` in the date example show in Listing 13.3. The following is the error line:

```
Today's date is <%= new java.util.Datex() %>
```

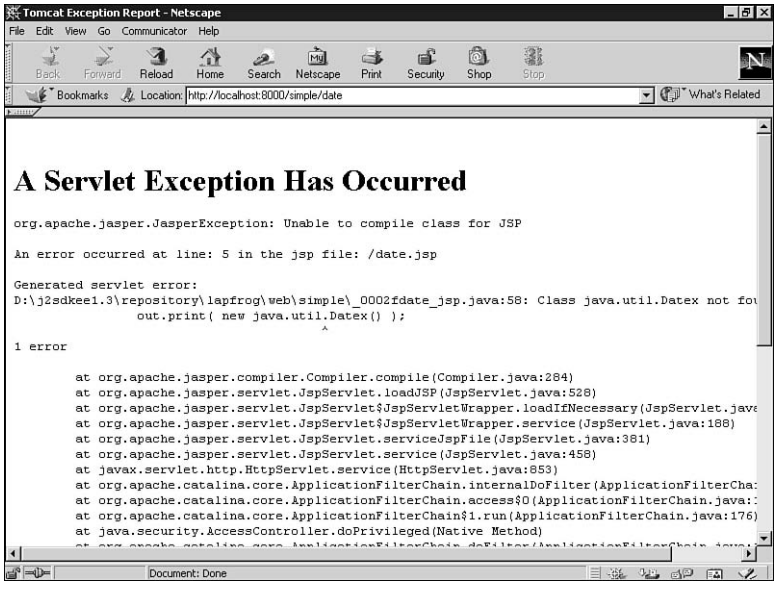


FIGURE 13.7

Browser showing JSP compilation error.

The information provided identifies the line in error in the JSP file and the corresponding line in error in the generated Java file. If you cannot determine the error from the JSP file, you will need to examine the generated file.

As stated earlier, the J2EE RI saves the generated Java file in the `repository` directory in the J2EE installation directory. The actual location is in a directory hierarchy named after the current workstation, the application name, and the Web application name. The filename is generated from the original JSP name.

In the example error, if the current host is ABC123, the file will be stored as

```
<J2EE home>\repository\ABC123\web\simple\0002fdate_jsp.java
```

The following code fragment shows the generated code containing the Java error:

```
application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="/date.jsp";from=(0,0);to=(4,20)]
out.write(
➥"<HTML>\r\n<TITLE>JSP Date Example</TITLE>\r\n<BODY>\r\n  <BIG>\r\n    Today's
date is ");

            // end
            // begin [file="/date.jsp";from=(4,23);to=(4,46)]
                out.print( new java.util.Datex() );
            // end
            // HTML // begin [file="/date.jsp";from=(4,48);to=(8,0)]
                out.write("\r\n  </BIG>\r\n</BODY>\r\n</HTML>\r\n");

            // end
```

As you can see, comments are inserted into the generated code to tie the Java code back to the original JSP code.

HTML Presentation Errors

The last category of error you can make with a JSP is to incorrectly define the HTML elements. These errors must be solved by looking at the HTML returned to the browser; most browsers have a menu option to let you view the HTML source for the page. After the HTML error is identified, you will have to relate this back to the original JSP file. Adding HTML comments to the JSP file can help you identify the location of the error if it is not readily apparent.

If no HTML data is returned to the browser, you have a serious problem with the JSP that is causing the generated servlet to fail without writing any data. You will need to examine your Web page very carefully for logic errors in the JSP elements; complex scriptlets are the most likely cause of the problem.

Your first step with a JSP that doesn't return HTML is to remove all of the JSP elements, leaving a plain HTML page, and ensure this is correctly returned to the browser. Gradually re-introduce JSP elements one at a time until the error reappears. Now you can correct the problem when you have identified where it occurs on the page.

Often, you will find that an HTML page is only partially complete. This usually indicates that the Java code has generated an exception part way through the Web page. Some Web servers (such as J2EE RI) will include the exception and a stack trace at the end of the incomplete page, others will write the error to a log file.

The JSP specification supports the concept of an error page that can be used to catch JSP exceptions, and this is discussed in more detail in the later sections on "The `page` Directive" and "Error Page Definition." Very briefly, an error page is displayed instead of the JSP when the JSP throws an exception. An error page can be a servlet, a JSP, or an HTML page. It is usual to define an error page containing debugging information during development and replace this with a "user friendly" version on a live system. The "Error Page Definition" section later in this chapter shows how to write a simple debugging error page for use during JSP development.

JSP Lifecycle Methods

The JSP equivalent of the servlet `init()` method is called `jspInit()` and can be defined to set up the JSP page. If present, `jspInit()` will be called by the server prior to the first request. Similarly, a method called `jspDestroy()` can be defined to deallocate resources used in the JSP page. The `jspDestroy()` method will be called when the page is taken out of service.

These methods must be defined inside a JSP declaration, as shown in the following:

```
<%!
  public void jspInit() {
    …
  }
  public void jspDestroy() {
    …
  }
%>
```

One of the problems with these lifecycle methods is that they are often used to initialize instance variables. Because JSPs are really servlets, the use of instance variables can cause problems with the multi-threading mechanisms used by the Web server for handling multiple page requests. If an instance variable is accessed concurrently from different threads, inconsistent values for the data may be obtained, leading to inconsistent behavior of the code. This is a well known problem that occurs in any multi-threaded Java program and is not confined to servlets.

Access to shared resources (like instance variables) on a JSP should be thread safe. Either all access to shared resources should be protected by `synchronized` blocks of code, or the page should implement the `SingleThreadModel` interface that forces the Web server to only run a single thread at a time on each servlet (as discussed on Day 12, "Servlets").

Because the servlet class is generated from your JSP code, you cannot implement the `SingleThreadModel` interface without support from the page translator. To specify translation requirements that affect the generated Java code for your Web page, you must add JSP directives to the page.

JSP Directives

Directives are used to define information about your page to the translator, they do not produce any HTML output. All directives have the following syntax:

```
<%@ directive [ attr="value" ] %>
```

where *`directive`* can be `page`, `include`, or `taglib`.

The `page` and `include` directives are described later, and the `taglib` directive is described tomorrow (Day 14) when Tag Libraries are studied in details.

The `include` Directive

You use the `include` directive to insert the contents of another file into the JSP. The included file can contain HTML or JSP tags or both. It is a useful mechanism for including the same page directives in all your JSPs or reusing small pieces of HTML to create common look and feel.

If the `include` file is itself a JSP, it is standard practice to use `.jsf` or `.jspf`, as suggested in the JSP specification, to indicate that the file contains a JSP fragment. These extensions show that the file is to be used in an `include` directive (and does not create a well-formed HTML page). "`.jsp`" should be reserved to refer to standalone JSPs.

Listing 13.4 shows a JSP with an `include` directive to add an HTML banner on the page. The banner is shown in Listing 13.5

LISTING 13.4 Full Text of `dateBanner.jsp`

```
1: <HTML>
2:   <HEAD>
3:     <TITLE>JSP Date Example with common banner</TITLE>
4:   </HEAD>
```

LISTING 13.4 Continued

```
 5:   <BODY>
 6:     <%@ include file="banner.html" %>
 7:     <BIG>
 8:       Today's date is
 9:       <%= new java.util.Date() %>
10:     </BIG>
11:   </BODY>
12: </HTML>
```

LISTING 13.5 Full Text of `banner.html`

```
1: <TABLE border="0" width="600" cellspacing="0" cellpadding="0">
2:   <TR>
3:     <TD width="350"><H1>Temporal Information  </H1> </TD>
4:      <TD align="right" width="250"><IMG src="clock.gif"> </TD>
5:    </TR>
6: </TABLE>
7: <BR>
```

Remember that you must add any include files into the Web application as well as the JSP file. In this example, you will need to add `banner.html` and the image file `clock.gif` to the Web Application.

The `page` Directive

Page directives are used to define page-dependent properties. You can have more than one page directive in the JSP. A page directive applies to the whole JSP, together with any files incorporated via the `include` directive. Table 13.2 defines a list of the more commonly used page directives.

TABLE 13.2 JSP Page Directives

Directive	*Example*	*Effect*
`info`	`<%@ page info="my first JSP Example" %>`	Defines text string that is placed in the `Servlet.getServletInfo()` method in the translated code

TABLE 13.2 Continued

Directive	*Example*	*Effect*
`import`	`<%@ page import=" java.math.*" %>`	A comma-separated list of package names to be imported for this JSP. The default import list is `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, and `javax.servlet.http.*`.
`isThreadSafe`	`<%@ page isThreadSafe="true" %>` `<%@ page isThreadSafe="false" %>`	If set to true, this indicates that this page can be run multi-threaded. This is the default, so you should ensure that access to shared objects (such as instance variables) is synchronized.
`errorPage`	`<%@ page errorPage="/agency/error.jsp" %>`	The client will be redirected to the specified URL when an exception occurs that is not caught by the current page.
`isErrorPage`	`<%@ page isErrorPage="true" %>` `<%@ page isErrorPage="false" %>`	Indicates whether this page is the target URL for an `errorPage` directive. If `true`, an implicit scripting variable called "`exception`" is defined and references the exception thrown in the source JSP. The default is `false`.

Using the Agency case study as an example, you can now study a JSP that needs to use page directives. The program in Listing 13.6 shows an example that displays the name of the Job Agency.

LISTING 13.6 Full Text of `name.jsp`

```
1: <%@page import="java.util.*, javax.naming.*, agency.*" %>
2: <%@page errorPage="errorPage.jsp" %>
3: <HTML>
4: <TITLE>Agency Name</TITLE>
5: <BODY>
6: <%
```

LISTING 13.6 Continued

```
 7:          InitialContext ic = null;
 8:          ic = new InitialContext();
 9:          AgencyHome agencyHome =
➥(AgencyHome)ic.lookup("java:comp/env/ejb/Agency");
10:          Agency agency = agencyHome.create();
11: %>
12: <H1><%= agency.getAgencyName() %> </H1>
13: </BODY>
14: </HTML>
```

Listing 13.6 uses the `agency` Session bean you developed on Day 5, "Session EJBs," to obtain the name of the agency. Because this JSP uses JNDI and EJB features, it must import the relevant Java packages by using a page directive. This example also uses an error page that is displayed if there is an uncaught exception on the page.

The JSP page does not catch the obvious `NamingException` and `EJBException` exceptions that can be thrown by the code in the scriptlet—the JSP error page is used for this purpose.

The error page contains JSP features that you have not yet encountered and will be shown later.

Deploying this example is a little more complex than the simple date example because it must include the additional class files required for the `agency` Session bean.

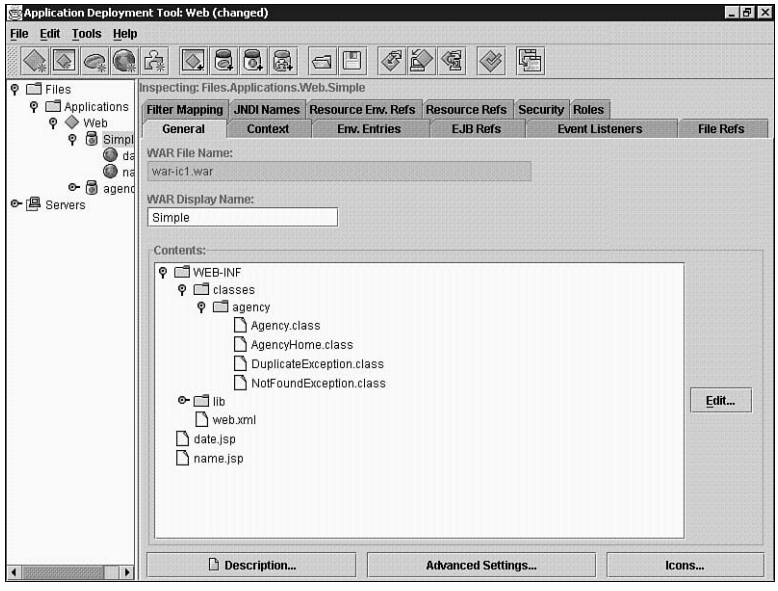
To deploy the example in Listing 13.6, perform the following steps:

1. Create a new Web Component in the `simple` `WAR` file you created earlier and use the wizard to define the information shown in step 2.
2. Add the following files for this Web component:
   - `name.jsp`
   - `errorPage.jsp`
   - `agency/Agency.class`
   - `agency/AgencyHome.class`
   - `agency/DuplicateException.class`
   - `agency/NotFoundException.class`

   Figure 13.8 shows the files in the `WAR` file for the `simple` Web application.

FIGURE 13.8

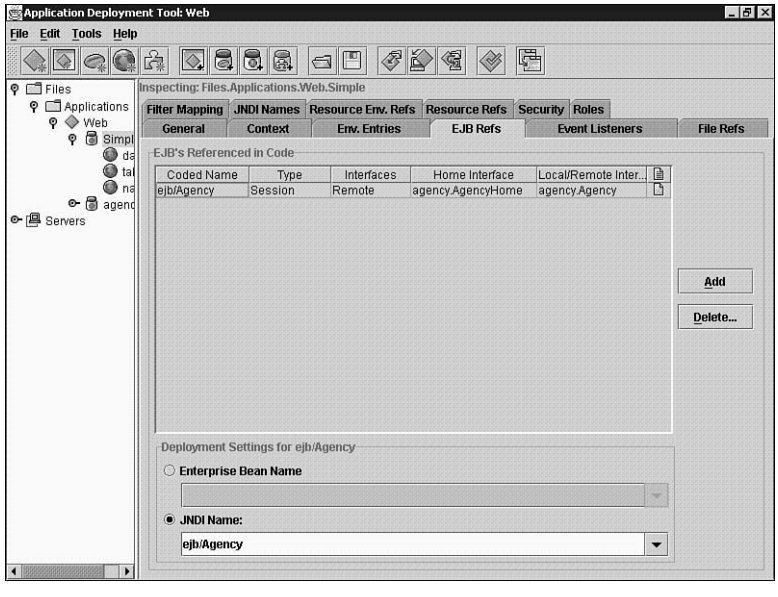
Component files for the `simple` *Web application.*



3. Select `name.jsp` as the JSP page.
4. Set the page alias as `/name`.
5. Add an EJB Reference for the agency EJB. Figure 13.9 shows the EJB Reference page with the required settings.

FIGURE 13.9

Agency EJB Reference for the simple Web *application.*



The new `name.jsp` Web page can now be deployed and accessed using the URL `http://localhost:8000/simple/name`.

> You must have previously deployed the `agency` application; otherwise, the Web interface will not be able to find the `agency` Session bean. Any version of the `agency` application will suffice, the one from Day 10, "Message-Driven Beans," is the one with most functionality.

Accessing HTTP Servlet Variables

The JSP pages you write are translated into servlets that process the HTTP `GET` and `POST` requests. The JSP code can access servlet information using implicit objects defined for each page. These implicit objects are pre-declared variables that you can reference from the Java code on your JSP. The most commonly used objects are shown in Table 13.3.

TABLE 13.3 JSP Implicit Objects

Reference Name	*Class*	*Description*
`config`	`javax.servlet.ServletConfig`	The servlet configuration information for the page
`request`	subclass of `javax.servlet.ServletRequest`	Request information for the current HTTP request
`session`	`javax.servlet.http.HttpSession`	The servlet session object for the client
`out`	`javax.servlet.jsp.JspWriter`	A subclass of `java.io.Writer` that is used to output text for inclusion on the Web page
`PageContext`	`javax.servlet.jsp.PageContext`	The JSP page context used primarily when implementing custom tags (see Day 14)
`Application`	`javax.servlet.ServletContext`	The context for all Web components in the same application

These implicit objects can be used on any JSP page. Using the date JSP shown in Listing 13.3 as an example, an alternative way of writing the date to the page is as follows:

```
<BIG>
      Today's date is <% out.print(new java.util.Date()); %>
</BIG>
```

Using HTTP Request Parameters

The next requirement for many JSP pages is to be able to use request parameters to configure the behavior of the page. Using the Agency case study as an example, you will develop a simple JSP to display the contents of a named database table.

The first step is to define a simple form to allow the user to select the table to display. Listing 13.7 shows a simple form encoded as a JSP.

LISTING 13.7 Full Text of `tableForm.jsp`

```
 1: <HTML>
 2: <TITLE>Agency Tables</TITLE>
 3: <BODY>
 4:   <FORM action=table>
 5:     Select a table to display:
 6:     <SELECT name=table>
 7:       <OPTION>Applicant
 8:       <OPTION>ApplicantSkill
 9:       <OPTION>Customer
10:       <OPTION>Job
11:       <OPTION>JobSKill
12:       <OPTION>Location
13:       <OPTION>Matched
14:       <OPTION>Skill
15:     </SELECT><P>
16:     <INPUT type=submit>
17:   </FORM>
18: </BODY>
19: </HTML>
```

Although the form in Listing 13.7 contains only HTML, it is convenient to treat it as a JSP. You can define an alias for this page and deploy it to the same Web resource location as the JSP that will display the table. This will simplify the application deployment and Web site management administration. The form component of the page is set to invoke the JSP called `table` in the current Web application (see line 4).

You will need to add this JSP to the `simple` Web application and define an alias of `/tableForm` to use it.

The actual JSP to display the table is shown in Listing 13.8.

LISTING 13.8 Full Text of `table.jsp`

```
1: <%@page import="java.util.*, javax.naming.*, agency.*" %>
2: <%@page errorPage="errorPage.jsp" %>
3: <% String table=request.getParameter("table"); %>
```

LISTING 13.8 Continued

```
 4: <HTML>
 5: <TITLE>Agency Table: <%= table %></TITLE>
 6: <BODY>
 7: <H1>Data for table <%= table %> </H1>
 8: <TABLE border=1>
 9: <%
10:         InitialContext ic = null;
11:         ic = new InitialContext();
12:         AgencyHome agencyHome =
➥(AgencyHome)ic.lookup("java:comp/env/ejb/Agency");
13:         Agency agency = agencyHome.create();
14:         Collection rows = agency.select(table);
15:         Iterator it = rows.iterator();
16:         while (it.hasNext()) {
17:               out.print("<TR>");
18:               String[] row = (String[])it.next();
19:               for (int i=0; i<row.length; i++) {
20:                   out.print("<TD>"+row[i]+"</TD>");
21:               }
22:               out.print("</TR>");
23:         }
24: %>
25: </TABLE>
26: </BODY>
27: </HTML>
```

The JSP in Listing 13.8 uses the implicit JSP variable called `request` to access the `ServletRequest` object (see line 3). The request parameters are retrieved by using the `getParameter()` method on the `request` object. The rest of this JSP simply creates an instance of the `agency` Session bean and uses the `select()` method to retrieve the table data. The data is formatted as an HTML table and output using the implicit `JspWriter` variable called `out`.

You will need to add this JSP to the `simple` web application and define an alias of `/table` to use it. Because the Web application file already contains an EJB Reference to the `agency` Session bean (from the `name.jsp` example in Listing 13.6), you do not need to define this EJB Reference again. If you deploy this Web component in a different Web application, you will need to add the EJB reference shown previously in Figure 13.9.

Simplifying JSP pages with JavaBeans

One of the problems with writing JSP pages is switching between the Java code and the HTML elements. It is easy to get confused and place syntax errors in the page that can

be difficult and time consuming to identify. Using JavaBeans on a JSP page can reduce the amount of embedded Java code that has to be written. JavaBeans also help to separate out the presentation and logic components of your application, allowing HTML developers to lay out the Web pages and the Java programmers to develop supporting JavaBeans.

What Is a JavaBean?

A bean is a self-contained, reusable software component. Beans are Java classes that are written to conform to a particular design convention (sometimes called an idiom). The rules for writing a JavaBean are as follows:

- A bean must have a no argument constructor.
- Beans can provide properties that allow customization of the bean. For each property, the bean must define *getter* and *setter* methods that retrieve or modify the bean property. For example, if a bean has a property called `name`, the bean class can define the methods `getName()` and `setName()`.
- The getter method must have no parameters and return an object of the type of the property. The setter method must take a single parameter of the type of the property and return a void. The following example shows a simple bean with a `String` property called `name` that can be queried and modified using the defined getter and setter methods.

```
public class NameBean {
  private String name;
  public void setName (String name) {
    this.name = name;
  }
  public String getName () {
    return name;
  }
}
```

If a bean property has only a getter method, it is read-only: a write-only method only has a setter method. A property is read/write if it has both getter and setter methods.

Beans can also define business methods to provide additional functionality above and beyond manipulating properties.

Defining a JavaBean

JavaBeans are defined on the JSP using the tag `<jsp:useBean>`. This tag creates an instance of a JavaBean and associates it with a name for use on the JSP.

```
<jsp:useBean id="<bean name>" class="<bean class>" scope="<scope>">
```

The bean name and class are defined by the `id` and `class` attributes for the `useBean` tag.

The `useBean` tag also requires a `scope` attribute that defines the scope of the bean reference. The possible scope values are as follows:

- `page` Only available on this page.
- `request` Available for this HTTP request (this page and any pages the request is forwarded to).
- `session` The duration of the client session (the bean can be used to pass information from one request to another). Session objects are accessed using the implicit object called `session`.
- `webcontext` The bean is added to the Web context and can be used by any other component in the Web application. The Web context is accessed using the implicit object called `application`.

The following code creates an instance of a bean of class `NameBean` for the current request and associates it with the name `myBean`.

```
<jsp:useBean id="myBean" class="NameBean" scope="request"/>
```

This bean has been defined using an empty JSP element because the bean is ready to use as soon as it has been defined. However, if the bean must be initialized, an alternate syntax is shown next and described fully in the "Initializing Beans" section later in the chapter.

```
<jsp:useBean id="myBean" class="NameBean" scope="request" >
  <jsp:setProperty name="myBean" property="name" value="winston"/>
</jsp:useBean>
```

Getting Bean Properties

Bean properties are retrieved using the `getProperty` element. This element requires a bean `name` (from the ID defined in the `useBean` tag) and `property` attribute, as shown in the following:

```
<jsp:getProperty name="<bean name>" property="<property name>" />
```

The value of the property is converted to a string and substituted on the Web page.

The following example shows how to retrieve the name property from the NameBean defined above and use it as a level 2 heading:

```
<H2><jsp:getProperty name="myBean" property="name"/></H2>
```

The getProperty tag has an empty body, so you should always define it shown in the example.

An alternative method for accessing a bean property is to use the bean name and get property method inside a JSP scripting element (such as an expression). The following code is an equivalent JSP rendering of the previous example:

```
<H2><%= myBean.getName() %></H2>
```

The advantage of this form of retrieving the property is that it less verbose when used as a value to an attribute for another JSP tag. The following shows how the name property can be used as the label on an HTML Submit button:

```
<INPUT type=submit value="Submit <%= myBean.getName() %>">
```

Setting Bean Properties

Bean properties are set using the setProperty element. This element requires a bean name (from the ID in the useBean element), a property, and a value attribute, as shown in the following:

```
<jsp:setProperty name="<bean name>"
➥property="<property name>" value="<expression>"/>
```

To set the name of the example NameBean to winston, you would use the following:

```
<jsp:setProperty name="myBean" property="name" value="winston"/>
```

As with getProperty, the bean method can be called explicitly from a Java scriptlet:

```
<% myBean.setName("winston"); %>
```

A useful feature of the setProperty tag is that bean properties can also be initialized from the HTTP request parameters. This is accomplished by using a param attribute rather than the value attribute:

```
<jsp:setProperty name="<bean name>" property="<property name>" param="<name>"/>
```

The value of the named parameter is used to set the appropriate bean property. To use a request parameter called name to set the NameBean property of the same name, you could use the following:

```
<jsp:setProperty name="myBean" property="name" param="name"/>
```

In fact, the `param` attribute can be omitted if the property name is the same as the request parameter name. So the previous example could have put more succinctly as follows:

```
<jsp:setProperty name="myBean" property="name"/>
```

A last form of the `setProperty` bean is employed when multiple parameters are used to initialize several bean properties. If the property name is set to `*`, all of the form request parameters are used to initialize bean properties with the same name:

```
<jsp:setProperty name="myBean" property="*"/>
```

The bean must define a property for every parameter in the HTTP request; otherwise, an error occurs.

Initializing Beans

Some beans require properties to be defined to initialize the bean. There is no mechanism for passing in initial values for properties in the `jsp:useBean` element, so a syntactic convention is used instead.

Conventionally, if a bean must have properties defined before it can be used on the Web page, the `jsp:useBean` is defined with an element body and the `jsp:setProperty` tags are defined in the `useBean` body to initialize the required properties.

For example, assuming that the simple `NameBean` example requires the name to be initialized, the following `useBean` syntax would be used:

```
<jsp:useBean id="myBean" class="NameBean" scope="request" >
  <jsp:setProperty name="myBean" property="name" value="winston"/>
</jsp:useBean>
```

Using a Bean with the Agency Case Study

The next example uses the Agency case study code and refactors the `name.jsp` Web page shown in Listing 13.6 to use a JavaBean. This time, you will use a bean to hide the complex JNDI lookup and type casting needed to access the `agency` Session EJB.

The new JSP page is shown in Listing 13.9.

LISTING 13.9 Full Text of `agencyName.jsp`

```
1: <HTML>
2: <TITLE>Agency Name</TITLE>
3: <BODY>
```

LISTING 13.9 Continued

```
4: <jsp:useBean id="agency" class="web.AgencyBean" scope="request" />
5: <H1><jsp:getProperty name="agency" property="agencyName"/></H1>
6: </BODY>
7: </HTML>
```

This is much simpler for a non-Java developer to work with. All of the code required to create the EJB using its JNDI name has been spirited away into a JavaBean of class `web.AgencyBean`. This bean does not have any properties but simply defines a large number of business methods whose only purpose is to delegate behavior to the underlying `agency` Session bean.

The full bean code is shown in Listing 13.10.

LISTING 13.10 Full Text of `web.AgencyBean.java`

```
 1: package web;
 2:
 3: import java.rmi.*;
 4: import java.util.* ;
 5: import javax.ejb.* ;
 6: import javax.naming.* ;
 7:
 8: import agency.*;
 9:
10: public class AgencyBean
11: {
12:     Agency agency;
13:
14:     public AgencyBean ()
➥throws NamingException, RemoteException, CreateException {
15:         InitialContext ic = null;
16:         ic = new InitialContext();
17:         AgencyHome agencyHome =
➥ (AgencyHome)ic.lookup("java:comp/env/ejb/Agency");
18:         agency = agencyHome.create();
19:     }
20:
21:     public String getAgencyName() throws RemoteException {
22:         return agency.getAgencyName();
23:     }
24:
25:     public Collection findAllApplicants() throws RemoteException {
26:         return agency.findAllApplicants();
27:     }
28:
```

LISTING 13.10 Continued

```
  29:     public void createApplicant(String login, String name, String email)
➥throws RemoteException, DuplicateException, CreateException{
  30:         agency.createApplicant(login,name,email);
  31:     }
  32:
  33:
  34:     public void deleteApplicant (String login)
➥throws RemoteException, NotFoundException{
  35:         agency.deleteApplicant(login);
  36:     }
  37:
  38:     public Collection findAllCustomers() throws RemoteException {
  39:         return agency.findAllCustomers();
  40:     }
  41:
  42:
43:     public void createCustomer(String login, String name, String email)
throws  RemoteException, DuplicateException, CreateException{
  44:         agency.createCustomer(login,name,email);
  45:     }
  46:
  47:     public void deleteCustomer (String login)
➥throws RemoteException, NotFoundException {
  48:         agency.deleteCustomer(login);
  49:     }
  50:
  51:     public Collection getLocations() throws RemoteException {
  52:         return agency.getLocations();
  53:     }
  54:
  55:     public String getLocationDescription(String name)
➥throws RemoteException, NotFoundException {
  56:         return agency.getLocationDescription(name);
  57:     }
  58:
  59:     public void updateLocation(String name, String description)
➥throws RemoteException, NotFoundException {
  60:         agency.updateLocation(name,description);
  61:     }
  62:
  63:     public void addLocation(String name, String description)
➥throws RemoteException, DuplicateException {
  64:         agency.addLocation(name,description);
  65:     }
  66:
  67:     public void removeLocation(String name)
➥throws RemoteException, NotFoundException {
  68:         agency.removeLocation(name);
```

LISTING 13.10 Continued

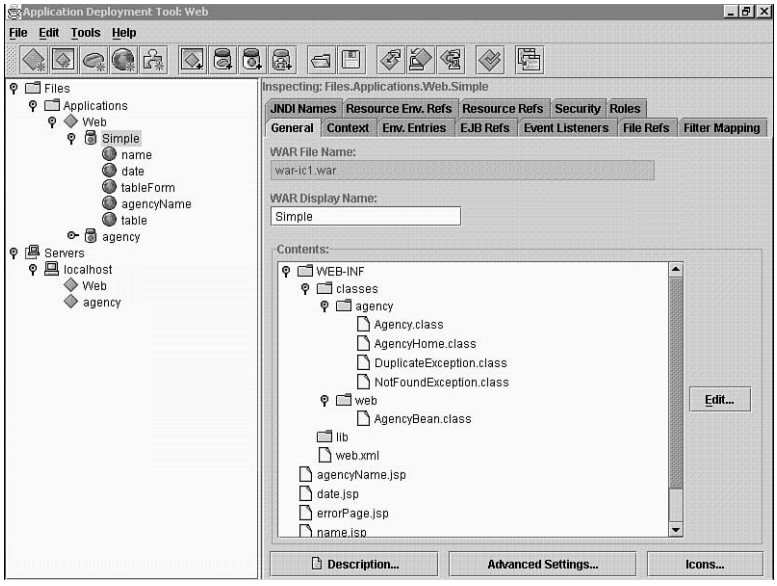
```
    }

    public Collection getSkills() throws RemoteException {
        return agency.getSkills();
    }

    public String getSkillDescription(String name)
➥throws RemoteException, NotFoundException {
        return agency.getSkillDescription(name);
    }

    public void updateSkill(String name, String description)
➥throws RemoteException, NotFoundException {
        agency.updateSkill(name,description);
    }

    public void addSkill(String name, String description)
➥throws RemoteException, DuplicateException {
        agency.addSkill(name,description);
    }

    public void removeSkill(String name)
➥throws RemoteException, NotFoundException {
        agency.removeSkill(name);
    }

    public List select(String table) throws RemoteException {
        return agency.select(table);
    }

}
```

The bean in Listing 13.10 is an example of an adapter (or wrapper) design pattern because it wraps around the `agency` session bean to simplify using the Session EJB on the JSP.

When deploying the `agencyName` example, if you add the new Web component to the simple Web application you will need to include the `web.agencyBean` class file, as well as the JSP. If you create a new Web application, you will also need to add the home and remote interfaces for the `agency` Session EJB. Figure 13.10 shows all the files in the simple Web application.

FIGURE 13.10

Component files for the `simple` Web application.



Adding a Web Interface to the Agency Case Study

Now that you have learned about most of the features of JavaServer Pages, you are in a position to add a Web front end to the Agency case study. You will only develop a simple, but functional, Web interface; it may not be too pretty, but it will work.

Designing a complete Web site is as much an art as a science, and what is considered good design by one developer may not be considered as good by another. You, too, will have your own views about what comprises a "good" Web site. Rather than get into the aesthetics of Web site design, this section will concentrate on applying the features of JSP in an effective manner.

Structure and Navigation

Designing a good Web site is a lot like designing good Java code. Both require careful logical thought and should incorporate good design principles and practices. A well-designed Java program is easier to maintain and enhance than a badly designed one. Similarly, a well-designed Web site is easy for users to navigate and use. A well-designed Web site will have proportionally less complaints and criticisms than a badly designed site.

In this section, you will only look at the logical structuring and layout of the Web pages and Web applications. The presentation of the page (its look and feel) is a very emotive subject, and what appeals to one user may not appeal to another. Furthermore, your Web page look and feel will almost certainly be constrained (or dictated) by department or corporate standards. The structuring of your Web site may well be guided by local standards or conventions, but even so, you can apply logic and some simple guidelines to improve the navigability and use of the Web site as a whole.

Consider structuring your Web site along the same lines as you store your files in directories. This way, you can use different Web applications to represent different functional areas of your systems, and use the Web application context and aliases to simplify navigation.

Some simple guidelines for good site layout are as follows:

- Group logically related functionality in a single Web application. By grouping related functionality into one application, you can reduce the learning time required for someone to maintain or link into your application because he or she only needs to study part of the site rather than the whole lot.
- Use Web component aliases to apply a structure to your Web pages. Consider using multiple aliases for a Web page so that multiple aliases can point to a single page, allowing for logical hierarchical naming conventions. The Agency case study used names to relate functionality by area as follows:

  ```
  /customer/create
  /customer/delete
  ```

  The aliases could equally as well have related logically similar functionality such as

  ```
  /delete/customer
  /delete/applicant
  ```

  Alternatively, both naming schemes could also have been provided. It all depends on how you view the inter-relationships between the various pages in your application.

  Whatever approach you choose, you should always use aliases and not the actual JSP names. This will allow you to restructure your Web site by adding, removing, or renaming JSP files without affecting hyperlinks from other pages.
- Ensure that no Web pages are dead ends (have no links to another page).
- Provide navigation links in the same place on every page.
- Ensure that every page has a link back to your site's home page and, if appropriate, a link to the top level page for this logical area of your system.

In the Agency case study, it has the following logical areas:

- Customers advertising jobs
- Applicants registering their locations and skills
- Administration of the location and skill lookup tables

Looking ahead to Day 15, "Security," when you will study the role of security within a J2EE application, you can structure your Web application to simplify the implementation of Web page authorization. You will do this by grouping all job and customer functionality under URLs starting with `customer` as follows:

```
/customer/advertise
/customer/createCustomer
/customer/deleteCustomer
/customer/updateCustomer
```

You will also store applicant functionality under pages with URLs starting with `applicant` as follows

```
/applicant/register
/applicant/createApplicant
/applicant/deleteApplicant
/applicant/updateApplicant
```

A Web site will typically have a main page called the *portal page* (also known as the home page or index page) that provides access to the functionality of the Web application. All clients will have access to the portal page, but they may have to login to use other features of the application. Because J2EE Web authorization is based on URL patterns, intelligent use of aliases for each Web page can ease the implementation of security to the Web application.

The Agency case study will have a portal page that will allow:

- An existing customer access his or her own details
- A new customer to be added to the system
- An existing applicant access to his or her own details
- A new applicant to register his or her details
- Access to administration features

You will start by studying some features of the main portal page for Job Agency application. The "Deploying the Case Study JSPs" section, later in this chapter, shows you how to deploy the code you will study.

Look and Feel

Using a consistent look and feel across Web pages is an important criteria principle. Consistent look and feel is supported by using the JSP `page include` directive and HTML Cascading Style Sheets (CSS).

JSP `page include` directives can be used to define common features for all pages, such as headers, footers, and navigation links. These directives can be used instead of server-side `include` directives implemented by some Web servers.

As an example of how to use `include` files is shown in the following code fragment:

```
<HTML>
<HEAD>
<TITLE>Agency Portal</TITLE>
<%@include file="header.jsf" %>
<!-- rest of page -->
```

Each Web page in the case study will include a common header file, shown in Listing 13.11.

LISTING 13.11 Full Text of `header.jsf`

```
1: <%@page errorPage="/agency/errorPage.jsp" %>
2: <LINK rel=stylesheet type="text/css" href="/agency/agency.css">
3: </HEAD>
4: <BODY>
5: <HR>
6: <jsp:useBean id="agency" class="web.AgencyBean" scope="request" />
7: <H1><jsp:getProperty name="agency" property="agencyName"/></H1>
8: <P>
```

Note that this file uses the `jsf` extension to show it is not a complete JSP. The header file defines an error page, a style sheet, and a common page heading that includes the job agency name.

By structuring the Web pages to include a common header file, you can be sure there will be a common look and feel to each page. The header file completes the page `<HEAD>` section and starts the page `<BODY>`. This isn't ideal, because the JSP designer has to know that the `include` file spans two logical components of the HTML page. But it does serve to show how useful `include` files can be.

The included header file also defines a bean called `agency` that can be used on the rest of the page to access the `agency` Session EJB. The level 1 page heading (`<H1>`) uses the agency name obtained from the `agency` bean.

If you haven't encountered HTML style sheets before, the simple one used for the agency is shown in Listing 13.12.

LISTING 13.12 Full Text of `agency.css`

```
1: H1, H2, H3 {font-family: sans-serif}
2: H1 {background-color: navy; color: white }
3: H2 {color: navy }
4: H3 {color: blue }
5: BODY, P, FORM, TABLE, TH, TD {font-family: sans-serif}
```

Without going into detail, all this style sheet does is define font styles and colors for use with the HTML tags on a Web page that links to this style sheet. Browser support for CSS is erratic, so you may not see all of the desired font changes with your browser. If in doubt about your users' Web browsers, you should not use stylesheets but embed font and color styles on the JSP itself instead.

Returning to the agency portal page, you can use the `agency` bean defined by the header to access the `agency` Session EJB functionality. When the user clicks the Show Customer button, the following JSP fragment presents the user with a list of customer names and invokes an `advertise.jsp` page (via its alias `customer/advertise`).

```
<FORM action="customer/advertise">
<TABLE>
<TR><TD>Select Customer</TD>
<TD><SELECT name="customer">
<% Iterator customers = agency.findAllCustomers().iterator(); %>
<% while (customers.hasNext()) {%>
  <OPTION><%=customers.next()%>
<% } %>
</SELECT>
</TD></TR>
<TR><TD colspan=2><input type=submit value="Show Customer"></TD></TR>
</TABLE>
</FORM>
```

The HTML select tag used on this form has to be encoded as a scriptlet. On Day 14, you will learn how Tag Libraries can be used to define custom tags that support iterative constructs like those required to support the customer select list.

Listing 13.13 shows all the code of the portal page for the `Agency` application with support for the customer functionality only.

LISTING 13.13 Full Text of `agency.jsp`

```
 1: <HTML>
 2: <HEAD>
 3: <TITLE>Agency Portal</TITLE>
 4: <%@include file="header.jsf" %>
 5: <%@page import="java.util.*" %>
 6: <H2>Customers</H2>
 7: <H3>Existing Customer</H3>
 8: <FORM action="customer/advertise">
 9: <TABLE>
10: <TR><TD>Select Customer</TD>
11: <TD><SELECT name="customer">
12: <% Iterator customers = agency.findAllCustomers().iterator(); %>
13: <% while (customers.hasNext()) {%>
14:   <OPTION><%=customers.next()%>
15: <% } %>
16: </SELECT>
17: </TD></TR>
18: <TR><TD colspan=2><input type=submit value="Show Customer"></TD></TR>
19: </TABLE>
20: </FORM>
21: <H3>Create Customer</H3>
22: <FORM action="customer/createCustomer">
23: <TABLE>
24: <TR>
25:   <TD>Login:</TD>
26:   <TD><INPUT type=text name=login></TD>
27: </TR>
28: <TR>
29:   <TD>Name:</TD>
30:   <TD><INPUT type=text name=name></TD>
31: </TR>
32: <TR>
33:   <TD>Email:</TD>
34:   <TD><INPUT type=text name=email></TD>
35: </TR>
36: <TR>
37:   <TD colspan=2><INPUT type=submit value="Create Customer"></TD>
38: </TR>
39: </TABLE>
40: </FORM>
41: <H2>Administration</H2>
42: <FORM action="admin/admin"><INPUT type=submit value="Administration
Form"></FORM>
43: <P>
44: <HR>
45: </BODY>
46: </HTML>
```

Your exercise to complete at the end of today's lesson will be to add support for registering applicants to this framework.

One last point to raise for the Agency Web pages is that of using a consistent navigation model for each page. JSP `include` files are an ideal method for including consistent hyperlinks in each Web page. The following code fragment shows how a standard footer with a navigation button for returning to the portal page is included at the end of each Web page, and Listing 13.14 shows the actual footer page.

```
<%@include file="footer.jsf" %>
</BODY>
</HTML>
```

LISTING 13.14 Full Text of `footer.jsf`

```
   1: <P>
   2: <HR>
   3: <P>
   4: <FORM>
   5: <INPUT type="button"
➥value="Return to Agency Menu" onClick='location="/agency/agency"'>
   6: </FORM>
   7: <BR>
```

If you consider the structure of jobs and applicants in the case study, you will realize that they have two common components:

- A location that will be a value from an HTML select list
- A set of skills (possibly) empty that must also be chosen from a list (in this case, an HTML select list that supports multiple options)

Good Java design practice would be to avoid duplicating the code by putting it in a separate method (or helper class) where the functionality can be reused. With JSP design, you can factor out the common code into a separate `include` file. The following code fragment shows how the JSP code for customer's advertised jobs uses `include` files:

```
<TABLE>
<TR><TD>Description:</TD>
      <TD><input type=text name=description
        value="<jsp:getProperty name="job" property="description"/>">
     </TD>
    </TR>
    <TR><TD>Location:</TD><TD>
      <% String location = job.getLocation(); %>
      <%@include file="location.jsf"%>
    </TD></TR>
```

```
    <TR><TD>Skills:</TD><TD>
      <% String[] skills = job.getSkills(); %>
      <%@include file="skills.jsf"%>
    </TD></TR>
</TABLE>
```

Each included file requires a variable to be defined before including the page (`location` and `skills`); this is the equivalent of passing a parameter into a method. This is not an ideal solution; in tomorrow's exercise, you will revisit this code and use tag libraries to develop a cleaner solution.

The previous code is part of the `advertise.jsp` file (see Listing 13.15). This JSP maintains customer details and jobs. It uses two JavaBeans as wrappers around the two Session EJBs called `CustomerBean` and `JobBean`. The `JobBean` wraps around the `advertiseJob` EJB that uses a compound parameter to identify each job (customer login and job reference). Creating this bean requires careful coding, as shown next. The list of customer jobs must be processed on the Web page in a loop implemented as a Java scriptlet, as shown in the following fragment:

```
<% String[] jobs = cust.getJobs(); %>
<jsp:useBean id="job" class="web.JobBean" scope="request">
  <jsp:setProperty name="job" property="customer" param="customer"/>
</jsp:useBean>
<% for (int i=0; i<jobs.length; i++) {%>
  <% job.setRef(jobs[i]); %>
  <H3><jsp:getProperty name="job" property="ref"/></H3>
…
<% } %>
```

Due to the way beans are defined on the page, this code has to be slightly clumsy. The bean is defined once before the Java loop and the customer part of the compound key are defined. Each time round the loop, the bean's `setRef()` method is called to set the next job reference; this method creates the appropriate `advertiseJob` Session EJB for use in the body of the loop. An alternative and much better approach is to use a custom tag, as shown on Day 14, "JSP Tag Libraries."

Listing 13.15 shows the entire advertise jobs page. The `include` files `location.jsf` and `skills.jsf` are included on the CD-ROM—not shown here.

LISTING 13.15 Full Text of `advertise.jsp`

```
1: <HTML><HEAD>
2: <TITLE>Advertise Customer Details</TITLE>
3: <%@include file="header.jsf" %>
4: <%@page import="java.util.*" %>
5: <jsp:useBean id="cust" class="web.CustomerBean" scope="request" >
```

LISTING 13.15 Continued

```
    6:   <jsp:setProperty name="cust" property="login" param="customer"/>
    7: </jsp:useBean>
    8: <H2>Customer details for:
➥<jsp:getProperty name="cust" property="login"/></H2>
    9: <FORM action="updateCustomer">
   10: <INPUT type=hidden name=login
➥value="<jsp:getProperty name="cust" property="login"/>">
   11: <TABLE>
   12:   <TR><TD>Login:</TD><TD>
➥<jsp:getProperty name="cust" property="login"/></TD></TR>
   13:   <TR><TD>Name:</TD><TD><input type=text name=name value="<jsp:getProperty
➥name="cust" property="name"/>"></TD></TR>
   14:   <TR><TD>Email:</TD><TD><input type=text name=email
➥value="<jsp:getProperty name="cust" property="email"/>"></TD></TR>
   15:   <% String[] address = cust.getAddress(); %>
   16:   <TR><TD>Address:</TD>
➥<TD><input type=text name=address value="<%=address[0]%>"></TD></TR>
   17:   <TR><TD>Address:</TD>
➥<TD><input type=text name=address value="<%=address[1]%>"></TD></TR>
   18: </TABLE>
   19: <INPUT type=submit value="Update Details">
   20: <INPUT type=reset>
   21: </FORM>
   22: <FORM action="deleteCustomer">
   23: <input type=hidden name=customer value="<%=cust.getName()%>">
   24: <input type=submit value="Delete Customer <%=cust.getName()%>"></TD></TR>
   25: </FORM>
   26: <H2>Jobs</H2>
   27: <% String[] jobs = cust.getJobs(); %>
   28: <jsp:useBean id="job" class="web.JobBean" scope="request">
   29:   <jsp:setProperty name="job" property="customer" param="customer"/>
   30: </jsp:useBean>
   31: <% for (int i=0; i<jobs.length; i++) {%>
   32:   <% job.setRef(jobs[i]); %>
   33:   <H3><jsp:getProperty name="job" property="ref"/></H3>
   34:   <FORM action="updateJob">
   35:     <TABLE>
   36:       <TR><TD>Description:</TD>
➥<TD><input type=text name=description value="<jsp:getProperty name="job"
prop➥erty="description"/>"></TD></TR>
   37:       <TR><TD>Location:</TD><TD>
   38:       <% String location = job.getLocation(); %>
   39:       <%@include file="location.jsf"%>
   40:       </TD></TR>
   41:       <TR><TD>Skills:</TD><TD>
   42:       <% String[] skills = job.getSkills(); %>
   43:       <%@include file="skills.jsf"%>
   44:       </TD></TR>
   45:     </TABLE>
```

LISTING 13.15 Continued

```
  46:     <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="job" property="customer"/>">
  47:     <INPUT type=hidden name="ref"
➥value="<jsp:getProperty name="job" property="ref"/>">
  48:     <INPUT type=submit value="Update Job">
  49:   </FORM>
  50:   <FORM action="deleteJob">
  51:     <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="job" property="customer"/>">
  52:     <INPUT type=hidden name="ref"
➥value="<jsp:getProperty name="job" property="ref"/>">
  53:     <INPUT type="submit"
➥value='Delete Job <jsp:getProperty name="job" property="ref"/>'>
  54:   </FORM>
  55: <% } %>
  56: <H2>Create New Job</H2>
  57: <FORM action="createJob">
  58: <TABLE>
  59: <TR>
  60:   <TD>Ref:</TD>
  61:   <TD><INPUT type=text name=ref></TD>
  62: </TR>
  63: <TR>
  64:   <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="cust" property="login"/>">
  65:   <TD colspan=2><INPUT type=submit value="Create Job"></TD>
  66: </TR>
  67: </TABLE>
  68: </FORM>
  69: <%@include file="footer.jsf" %>
  70: </BODY>
  71: </HTML>
```

To complete the customer functionality, the Web interface requires separate pages for

- Creating a new customer
- Deleting a customer
- Updating a customer
- Creating a new job
- Deleting a job
- Updating a job

Rather than reproduce all of these pages, the page for updating a customer is shown in Listing 13.16 (the remainder can be examined on the CD-ROM). Each page simply creates the appropriate agency, customer, or job bean using the request parameters, and calls the necessary business method.

LISTING 13.16 Full Text of `updateCustomer.jsp`

```
 1: <HTML>
 2: <HEAD>
 3: <TITLE>Update Customer</TITLE>
 4: <%@include file="header.jsf" %>
 5: <jsp:useBean id="cust" class="web.CustomerBean" scope="request" >
 6:   <jsp:setProperty name="cust" property="login" param="login"/>
 7: </jsp:useBean>
 8: <% cust.updateDetails(request.getParameter("name"),
➥request.getParameter("email"),request.getParameterValues("address")); %>
 9: <H3>Updated <jsp:getProperty name="cust" property="login"/>
➥Successfully</H3>
10: <%@include file="footer.jsf" %>
11: </BODY>
12: </HTML>
```

Error Page Definition

The very last file to examine for the case study is the error page file shown in Listing 13.17.

LISTING 13.17 Full Text of `errorPage.jsp`

```
 1: <%@ page isErrorPage="true" %>
 2: <%@page import="java.util.*, java.io.* " %>
 3: <HTML>
 4: <HEAD>
 5: <TITLE>Agency Error Page</TITLE>
 6: </HEAD>
 7: <BODY>
 8: <H1>Agency Error Page</H1>
 9: <H2>There has been an error in processing your request.</H2>
10: The following information describes the error:
11: <H3>Request Parameters</H3>
12: <TABLE border=1>
13: <%
14: Enumeration params = request.getParameterNames();
15: while (params.hasMoreElements()) {
16:   String name = (String)params.nextElement();
17:   out.println("<TR><TD>"+name+"</TD>");
18:   String[] values = request.getParameterValues(name);
19:   for (int i=0; i<values.length; i++) {
20:     out.println("<TD>"+values[i]+"</TD>");
21:   }
22:   out.println("</TR>");
23: }
24: %>
```

LISTING 13.17 Continued

```
25: </TABLE>
26: <H3>Request Attributes</H3>
27: <TABLE border=1>
28: <%
29: Enumeration attrs = request.getAttributeNames();
30: while (attrs.hasMoreElements()) {
31:   String name = (String)attrs.nextElement();
32:   out.println("<TR><TD>"+name+"</TD>");
33:   out.println("<TD>"+request.getAttribute(name)+"</TD>");
34:   out.println("</TR>");
35: }
36: %>
37: </TABLE>
38: <H3>Session Attributes</H3>
39: <TABLE border=1>
40: <%
41: Enumeration sess = session.getAttributeNames();
42: while (sess.hasMoreElements()) {
43:   String name = (String)sess.nextElement();
44:   out.println("<TR><TD>"+name+"</TD>");
45:   out.println("<TD>"+session.getAttribute(name)+"</TD>");
46:   out.println("</TR>");
47: }
48: %>
49: </TABLE>
50: <H3>Exception</H3>
51: <%=exception%> <%=exception.getMessage()%>
52: <H3>Stack Trace</H3>
53: <%
54: StringWriter buf = new StringWriter();
55: PrintWriter sout = new PrintWriter(buf);
56: exception.printStackTrace(sout);
57: out.println(buf.toString());
58: %>
59: </BODY>
60: </HTML>
```

This error page is designed for use during development. When an exception occurs, this page displays information about the exception and various Java variables derived from the request, servlet, and page contexts.

At the top of Listing 13.17 is the line

```
<%@ page isErrorPage="true" %>
```

This tells the translation phase to include a variable called `exception` that refers to the exception that caused the page error. This exception is used to display a stack trace on the error page. The first line of the stack trace will identify

- The name of the generated servlet
- The line number where the exception occurred
- The exception that was thrown
- A brief description of the error

This information can be used to trace back the error to the original JSP file by using the Java code listing for the generated servlet.

For a fully-developed and deployed application, it would be better for the error page to display a user-friendly error message and report the error (perhaps via JavaMail) to an administrator.

The Agency case study error pages are designed to illustrate the principles involved in error reporting and are not necessarily an example of best practice.

Deploying the Case Study JSPs

In today's lesson, you have used a large number of files to create the Web interface to the job `agency` Session beans. Building and deploying the Web application is relatively straightforward if you perform the following steps:

1. Start up `deploytool` and create a new Web component and add it to a new `WAR` file called `web`.
2. Add the following JSP files from the `src/jsp` directory to this war file:
    - `admin.jsp`
    - `advertise.jsp`
    - `agency.css`
    - `agency.jsp`
    - `createCustomer.jsp`
    - `createJob.jsp`
    - `createLocation.jsp`
    - `createSkill.jsp`
    - `deletCustomer.jsp`
    - `deleteJob.jsp`
    - `deleteLocation.jsp`
    - `deleteSkill.jsp`
    - `errorPage.jsp`
    - `footer.jsf`

- header.jsf
- location.jsf
- modifyLocation.jsp
- modifySkill.jsp
- skills.jsf
- updateCustomer.jsp
- updateJob.jsp
- updateLocation.jsp
- updateSkill.jsp

3. Add the following class files from the web directory to this WAR file:
   - AgencyBean.class
   - CustomerBean.class
   - JobBean.class
4. Add the following class files from the agency directory to this WAR file:
   - Advertise.class
   - AdvertiseHome.class
   - AdvertiseJob.class
   - AdvertiseJobHome.class
   - Agency.class
   - AgencyHome.class
   - DuplicateException.class
   - NotFoundException.class
   - Register.class
   - RegisterHome.class
5. Click Next, select JSP for the EJB component, and the click Next.
6. Set the JSP Filename to agency.jsp and accept the default Web component name of admin.
7. Click Next twice and add a page alias of /agency.
8. Click Next four times to get to the EJB References page. Add the EJB references shown in Table 13.4 (they are all Session beans with a remote interface):

TABLE 13.4 Case Study Web Application EJB References

Coded name	*Home i/f*	*Remote i/f*	*JNDI name*
ejb/agency	agency.AgencyHome	agency.Agency	ejb/agency
ejb/advertise	agency.AdvertiseHome	agency.Advertise	ejb/advertise
ejb/advertiseJob	agency.AdvertiseJobHome	agency.AdvertiseJob	ejb/advertiseJob

9. Click Finish.
10. Now you will need to create Web components in the same WAR file for every other JSP. You have added all of the required files, so all you need to do is define the JSP page and alias. Table 13.5 lists all the pages you need to define, including the admin JSP page you just added in step 6.

TABLE 13.5 Case Study Web Application JSP Components

JSP Filename	*Alias*
advertise.jsp	/customer/advertise
admin.jsp	/admin/admin
createCustomer.jsp	/customer/createCustomer
createJob.jsp	/customer/createJob
createLocation.jsp	/admin/createLocation
createSkill.jsp	/admin/createSkill
deletCustomer.jsp	/customer/deleteCustomer
deleteJob.jsp	/customer/deleteJob
deleteLocation.jsp	/admin/deleteLocation
deleteSkill.jsp	/admin/deleteSkill
errorPage.jsp	/errorPage
updateCustomer.jsp	/customer/updateCustomer
updateJob.jsp	/customer/updateJob
updateLocation.jsp	/admin/updateLocation
updateSkill.jsp	/admin/updateSkill

11. When you have added all of the Web components, you can deploy the Web application with Tools, Deploy. You do not need to return the client JAR file, so click Next twice.
12. On the WAR Context Root screen, enter a root context for your Web application of agency.

13. Click Finish and, if you've entered everything correctly, you will have deployed your case study Web interface.

You must have previously deployed the Agency application with which the Web interface can communicate. Any version of the Agency case study from Day 5, "Session EJBs," Day 6, "Entity EJBs," Day 10, "Message-Driven Beans," or Day 11, "JavaMail" will suffice.

You can access the case study application using `http://localhost:8000/agency/agency`.

Comparing JSP with Servlets

As you have seen JSPs have several advantages over servlets. They

- Are quicker to write and develop
- Focus on the page layout and delegate Java logic to supporting JavaBeans and custom tags (as will be discussed on Day 14)
- Differentiate the Web page presentation (HTML) from the underlying logic (Java)
- Can be written by non-Java–aware developers provided suitable supporting beans and tag libraries are available
- Support a standard error reporting mechanism using the error page directive

However, JSPs do have some downsides:

- Error identification and correction is complicated by the translate and compile lifecycle.
- Large volumes of embedded Java scriptlets can reduce the maintainability of the page.

In general, you should use JSPs with JavaBeans and custom tag libraries whenever possible. The speed of development and the quick turnaround on look and feel or simple functional changes that is possible with JSPs is a major advantage in modern Web-based applications.

Consider using servlets only when the Java code is complex or needs to be "hand crafted" for efficiency.

One common approach for supporting complex Java requirements is to use a servlet for the Java code and a supporting JSP for the presentation. The Java servlet accesses the information and stores it in the session context. The servlet then forwards the HTTP request to the JSP, which retrieves the data from the session context and presents it back to the client as HTML. This approach still maintains the differentiation between presentation and logic.

An additional advantage of this servlet/JSP approach is that the servlet can use one of several JSPs to present the data according to the type of client making the request. You can define one JSP for HTML clients (PCs) and another for WML clients (mobile phones). In the future, you can add additional clients for, say, XML clients or a presentation language than has not yet been defined. All of these different presentation requirements can be supported by a single servlet. Any future changes to the underlying business rules are only made once in the logic of the servlet, with any necessary changes in the presentation being made in the appropriate JSPs.

Summary

Today, you have looked at using JavaServer Pages as a means of developing Web-based J2EE applications. Unlike servlets, JSPs allow you to develop your Web pages in HTML with embedded Java code when dynamic elements are required.

JSP pages are translated into Java servlets and compiled before they are used to service client requests. To the browser, a JSP is no different to any other Web page.

JSP defines three elements to supplement the standard HTML tags:

- Directives that are used to pass information to the page translation phase
- Scripts that define Java code used to embed dynamic data in the page
- Actions that define JSP tags used to support JavaBeans

JavaBeans are used to encapsulate Java functionality to remove some Java script code from the Web page. Beans help separate the role of Java developer from that of HTML Web developer. Beans are classes that have a no argument constructor and properties. Bean properties have names, are queried using *getter* methods, and are updated using *setter* methods.

In the next lesson, you will study defining your own custom tags as part of a tag library. Tag libraries will support complex page features, such as the iterative processing of dynamic data (such as a list of customers). With carefully constructed tag libraries, a JSP page can be written without using any Java code at all.

Q&A

Q What are the three types of errors that can occur on a JSP?

A Translation, compilation, and HTML syntax errors.

Q What are the three types of JSP elements?

A Directives, scripts, and actions.

Q What are the three types of Java scripts?

A Declarations, expressions, and scriptlets.

Q What are the three JSP actions that support the use of JavaBeans?

A `<jsp:useBean>`, `<jsp:setProperty>`, and `<jsp:getProperty>`.

Q What sort of constructor must a JavaBean have?

A It must have a no argument constructor.

Exercise

Using the JSP code shown in today's lesson, provide the register applicant functionality for the Web interface to the agency case study.

You will need to perform the following steps. Build and deploy your application as you complete each step.

1. Add a `createApplicant.jsp` page with the alias `/applicant/createApplicant` to support the create applicant functionality on the `agency.jsp` page. Model your solution on the `createCustomer.jsp` page already provided.
2. Using `advertise.jsp` as a guide, develop a Web page called `register.jsp` to handle client registration. The links to invoke this page from the `agency.jsp` portal page have already been provided.
3. Add the ability to update the `applicant` details and code an `updateApplicant.jsp` page that is accessed from a form on the `register.jsp` page and has the alias `/applicant/updateApplicant`.
4. Add functionality to the `register.jsp` page to delete an applicant and write the `deleteApplicant.jsp` page (alias `/applicant/deleteApplicant`).

WEEK 2

DAY 14

JSP Tag Libraries

In the previous two day's chapters, you have learned about J2EE Web applications written using servlets and JSPs. You have seen how servlets are most useful when complex Java programming is needed and JSPs are easier to use when the generated Web page requires large amounts of HTML (or JavaScript). However, JSPs are essentially static in nature and require the developer to write Java code, in the form of scriptlets, to support complex features.

JSP Tag Libraries (TagLibs) are a natural extension to the JSP tag syntax. TagLibs are custom tags that are written in Java but interact with the processing of the tags on the JSP page. In this chapter, you will learn how to

- Write simple custom tags
- Understand the XML deployment descriptor for Tag Libraries
- Deploy your tags with the J2EE RI
- Use attributes to extend JSP pages
- Write co-operating tags to support more complex functionality, such as iteration
- Use the JSP Standard Tag Library (JSPTL)

The Role of Tag Libraries

JSPs enable you to develop Web-based applications using Java without being a Java expert. By using well designed JavaBeans, it is possible to work almost entirely with HTML and JSP tags. However, as you saw in yesterday's work, you still have to use Java scriptlets to realize the full power of JSPs.

Tag Libraries extend the JSP philosophy further still so that it is possible for you to write most of your Web pages without using Java code.

This can utilized to good effect in development teams. By separating out the Java code into custom tags, a development team can utilize their individual skills more effectively. Java programmers are used to develop the business logic in custom tags, while HTML/JSP developers can focus on developing the presentation logic and look-and-feel of the Web pages.

Tag Libraries support custom JSP tags that can

- Be customized by using attributes passed from the calling page
- Access the objects available to JSP pages
- Communicate with each other
- Be used in a hierarchical manner to support complex features, such as iteration

Tag libraries were introduced with JSP 1.1 (1999) and the supporting Java classes were slightly revised in JSP 1.2 (2001). Since the introduction of JSPs the user community has quickly adopted them. Popular Web servers supporting Tag Libraries include the following:

- Apache Jakarta (open source)
- BEA Web Logic
- IBM Web Sphere
- iPlanet Applications Server

Most Web servers supporting JSP now include some standard Tag Libraries. However, these Tag Libraries are proprietary to each Web server, leading to Web applications that can only be deployed on a specific manufacturers server. In 2001, the Java Community Process (JCP) proposed a JSP Standard Tag Library (JSPTL) that, at the time of writing this book, was still being evaluated. The proposed standard library is discussed at the end of today's work and is available from the Apache Jakarta Project at `http://jakarta.apache.org/taglibs`.

Developing a Simple Custom Tag

A custom tag is made up of two components:

- A Java class file that implements the tag
- An entry in a Tag Library Descriptor (TLD) file that defines the tag's name, its implementing Java class, and additional information necessary to deploy and use the tag

Using a custom tag requires a reference to the Tag Library Descriptor (TLD) at the start of the JSP. Multiple Tag Libraries can be used on the same Web page. After the TLD has been referenced, the custom tag from the TLD can be used like any other JSP tag.

Using a Simple Tag

To start learning to write and use TagLibs, you will implement a very simple custom tag that writes "Hello World" onto your Web page. This isn't a good use of a custom tag, but it will help you understand the principles involved and guide you through the deployment process. After this simple example, you will look at how to use custom tags to remove most of the Java scriptlets from your JSP applications.

The example in Listing 14.1 shows a simple tag that inserts `Hello World` on the JSP page.

LISTING 14.1 Full Text of `hello.jsp`

```
1: <%@ taglib uri="/demo" prefix="demo" %>
2: <HTML>
3: <HEAD>
4: <TITLE>Tag Library Hello Demo</TITLE>
5: <BODY>
6:   <demo:hello/>
7: </BODY>
8: </HTML>
```

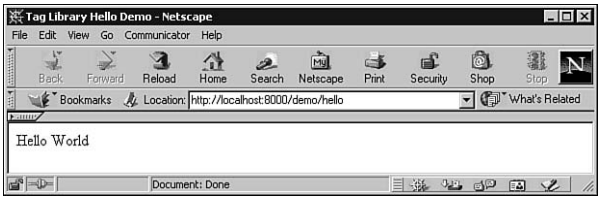
Like XML (and unlike HTML), the JSP tags are case sensitive. JSP tags are also XML compliant and must have all parameter values enclosed in single or double quotes.

The very first line of the JSP in Listing 14.1 defines the Tag Library and associates the tags in the library with the `demo` prefix.

The tag itself is used in line 6, and its name is `hello`. The resulting Web page is shown in Figure 14.1.

FIGURE 14.1

Browsing the "Hello World" tag.


Tag Library Hello Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8000/demo/hello
What's Related
Hello World
Document: Done

The Tag Library Descriptor (TLD)

TLD files map the tag name used in the JSP page onto the Java class that implements the tag. A TLD file is written in XML and an example is given in Listing 14.2, which shows the TLD file for the "Hello World" tag.

LISTING 14.2 Full Text of `demo.tld`

```
 1: <?xml version="1.0" encoding="ISO-8859-1" ?>
 2: <!DOCTYPE taglib PUBLIC
 3:         "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 4:         "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
 5: <taglib>
 6:   <tlib-version>1.0</tlib-version>
 7:   <jsp-version>1.2</jsp-version>
 8:   <short-name>demo</short-name>
 9:   <tag>
10:     <name>hello</name>
11:     <tag-class>demo.HelloTag</tag-class>
12:     <body-content>empty</body-content>
13:   </tag>
14: </taglib>
```

At the present time, the J2EE RI from Sun Microsystems does not include any support for generating this file from within `deploytool`. You will have to write your TLD files by hand and include them in your Web Application.

To write a TLD, you will need to be familiar with XML. If you are new to XML, this might be a good time to have a quick look at Appendix C, "An Overview of XML," before continuing to learn how TLD files work. The use of XML in enterprise applications is also discussed on Day 16, "Integrating XML with J2EE."

Every TLD file begins with a prolog that

- Defines the XML version used in the TLD
- Defines the version of the specification to which the TLD conforms

The prolog, shown in the following lines should always be included at the start of your TLD as it is used by tools processing the document to validate the XML structure:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC
       "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
       "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

The first line defines the TLD as using version 1.0 of the XML standard. The second line defines the TLD as conforming to version 1.2 of the TLD Specification from Sun Microsystems.

The Tag Library itself is defined by a `<taglib>` element. The `<taglib>` element specifies the Tag Library version (1.0) and JSP version (1.2) to which this Tag Library conforms as follows:

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>demo</short-name>
...
</taglib>
```

The third element in the `<taglib>` section is an optional `<short-name>` entry that should be the preferred prefix name for your library. The actual library prefix is defined in the `<%@ taglib>` directive on the JSP page. The prefix in the `<%@ taglib>` directive can be different from the `<short-name>` entry. The only reason for changing the prefix on the JSP page is where two separate Tag Libraries suggest the same prefix name, and the JSP page designer has to resolve the conflict.

After the Tag Library information is defined, the rest of the TLD is used to specify the custom tags in the library.

Every custom tag potentially has three parts:

- The starting tag, such as `<demo:hello>`
- The ending tag, such as `</demo:hello>`
- The tag's body, which is the text between the start and end tags

A tag that does not have a body is called an *empty tag* and is usually written as `<demo:hello/>`.

In the TLD, a <tag> tag defines a custom tag. Several custom tags can be defined in a single library. The minimal requirements for each custom tag are as follows:

- The tag name to be used on the Web page
- The Java class name that implements the tag
- The type of tag body

In the following example, the tag class is demo.HelloTag, and the tag name is hello:

```
<tag>
  <name>hello</name>
  <tag-class>demo.HelloTag</tag-class>
  <body-content>empty</body-content>
</tag>
```

In this case, the <body-content> tag defines the tag to have an empty body. The possible values for the <body-content> tag are shown in Table 14.1.

TABLE 14.1 Tag Body Contents

Tag	*Description*
empty	The tag body must be empty. The page translation will fail if the developer puts any text between the start and end tags.
JSP	The tag contains JSP data and will normally be processed in the same manner as other parts of the page.
tagdependent	The text between the start and end tags will be processed by the Java class and will not be interpreted as JSP tagged content.

The <body-content> type must be compatible with the way the Java code processes the tag, as discussed in the next section.

Custom Java Tags

You define custom tags by using Java classes that implement one of the javax.servlet.jsp.tagext interfaces shown in Table 14.2 .

TABLE 14.2 Interfaces for Custom Tags

Java Interface	*Description*
Tag	Implement this interface for simple tags that do not need to process the tag body.
IterationTag	Implement this interface for tags that need to process the body text more than once to implement an iterative loop.
BodyTag	Implement this interface for tags that need to process the text in the tag body.

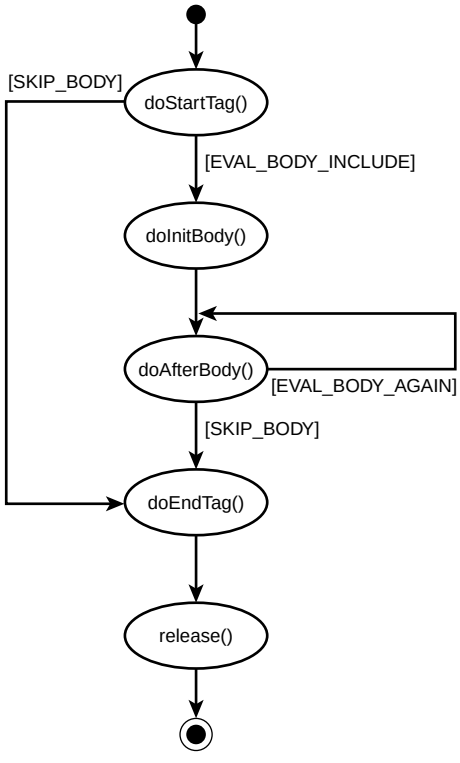
Implementing these interfaces requires that you define several methods to manage the lifecycle of the custom tag. To simplify custom tag development, two support classes are provided in the `javax.servlet.jsp.tagext` package. These classes provide default behavior for each of the required interface methods, and you will simply override the methods you require. Table 14.3 shows the two supporting classes.

TABLE 14.3 Support Classes for Custom Tags

java Class	*Description*
`TagSupport implements Tag`	Extend this class for tags that do not have a body or do not interact with the tag body.
`BodyTagSupport implements Tag, IterationTag, BodyTag`	Extend this class when the tag body must be processed as part of the tag, such as an iterative tag or a tag that interprets the body in some way.

The `Tag`, `TagIteration`, and `TagSupport` interfaces in the `javax.servlet.jsp.tagext` package define several methods that control the processing of the custom tag. Figure 14.2 summarizes the lifecycle of a Tag Library object.

FIGURE 14.2
The Custom Tag lifecycle.



The `doStartTag()` Method

The `doStartTag()` method is called once when the start tag is processed. This method must return an `int` value that tells the JSP how to process the tag body. The returned value must be one of the following:

- `Tag.SKIP_BODY` The tag body must be ignored. The TLD should define the `<body-content>` tag as `empty`.
- `Tag.EVAL_BODY_INCLUDE` The body tag must be evaluated and included in the JSP page. The TLD should define the `<body-content>` tag as `JSP` for tags that extend `TagSupport`, or `JSP` or `tagdependent` for tags that extend `BodyTagSupport`.

The `doEndTag()` Method

The `doEndTag()` method is called once when the end tag is processed. This method must return an `int` value indicating how the remainder of the JSP page should be processed:

- `Tag.EVAL_PAGE` Evaluate the rest of the page.
- `Tag.SKIP_PAGE` Stop processing the page after this tag.

Where a tag has an empty body, the `doEndTag()` method is still called after the `doStartTag()` method.

The `release()` Method

The `release()` method is called once when the JSP has finished using the tag and is used to allow the tag to release any resources it may have acquired. This method's return type is `void`.

The `doAfterBody()` Method

The `doAfterBody()` method is called after the tag body has been processed and before the `doEndTag()` method is called. This method is only called for classes that implement `IterationTag` or `BodyTag` (those that extend `BodyTagSupport`). It must return one of the following values to the JSP page indicating how the tag body should be processed:

- `IterationTag.EVAL_BODY_AGAIN` This value is used to inform the page that the tag body should be processed once more. The JSP processing will read and process the tag body and call the `doAfterBody()` method once more after the body has been processed again.

When returning the EVAL_BODY_AGAIN result, this method will typically change some value so that when the tag body is processed again, different output is written to the Web page. A simple example would be a tag that executes a database query in the doStartTag() method and reads the next row of the result set in this method returning EVAL_BODY_AGAIN until the end of the result set is reached when SKIP_BODY is returned.

- Tag.SKIP_BODY This value marks the end of the processing of the tag body.

The doInitBody() Method

The doInitBody() method is called once after the doStartTag() method but before the tag body is processed. doInitBody() is only used in tags that implement the BodyTag interface (those that extend BodyTagSupport). This method is not called if the doStartTag() method returns SKIP_BODY. The doInitBody() method returns void.

The "Hello World" Custom Tag

The example Hello World tag shown in Listing 14.1 does not have a body, so it need only implement the Tag interface, which is best achieved by extending the TagSupport class and overriding the required methods, as shown in Listing 14.3.

LISTING 14.3 Full Text of HelloTag.java

```
 1: package demo;
 2:
 3: import javax.servlet.jsp.*;
 4: import javax.servlet.jsp.tagext.*;
 5:
 6: public class HelloTag extends TagSupport {
 7:    public int doStartTag() throws JspException {
 8:       try {
 9:          pageContext.getOut().print("Hello World");
10:       } catch (Exception ex) {
11:          throw new JspTagException("HelloTag: "+ex);
12:       }
13:       return SKIP_BODY;
14:    }
15:
16:     public int doEndTag() {
17:       return EVAL_PAGE;
18:    }
19:
20:    public void release() {
21:    }
22: }
```

What little work this tag does is done in the `doStartTag()` method. On line 9, the current output stream is obtained from the page context (an instance variable called `pageContext` of class `javax.servlet.jsp.tagext.PageContext`) and used to print the `"Hello World"` string:

```
pageContext.getOut().print("Hello World");
```

With all the code for the custom tag defined, you can now deploy the `Hello World` example as described in the next section.

Deploying a Tag Library Web Application

To deploy a Web Application that uses a Tag Library, you will follow the same basic steps as for a normal Web Application, but you will need to include the Tag Library information at the relevant points:

1. Start up the J2EE RI `deploytool`.
2. Either create a new Application or select an existing application to use for the Tag Library example.
3. Create a new Web Component and add it to a new WAR file called `Demo`.
4. Include the files for the JSP page (`hello.jsp`) and the Java class file (`demo.HelloTag.class`) (the class files will be added to the `WEB-INF/classes` sub-directory).
5. Include the Tag Library TLD file (`demo.tld`), and this will be added to the `WEB-INF` directory for your application. At this point, your deployment screen will look like Figure 14.3.
6. Continue deploying the Web Application, click `Next` twice to move on to the General Properties screen and set the main JSP page to `hello.jsp`.
7. Click Next twice more to move on to the Aliases screen and add a `/hello` alias for the JSP page.
8. Step through the next seven configuration screens until the File References screen is displayed. Define an entry for your TLD in the JSP Tag Libraries section (half way down the screen). Use the TLD shortcut name you used in the `hellp.jsp` file (`/demo`) and map this onto the actual TLD (`/demo.tld`)—note that you need only give the filename as `deploytool` assumes the TLD is in the WEB-INF directory. At this point, your screen will look like the one in Figure 14.4.
9. Click the Finish button.

FIGURE 14.3

Creating the Tag Library Web Application.

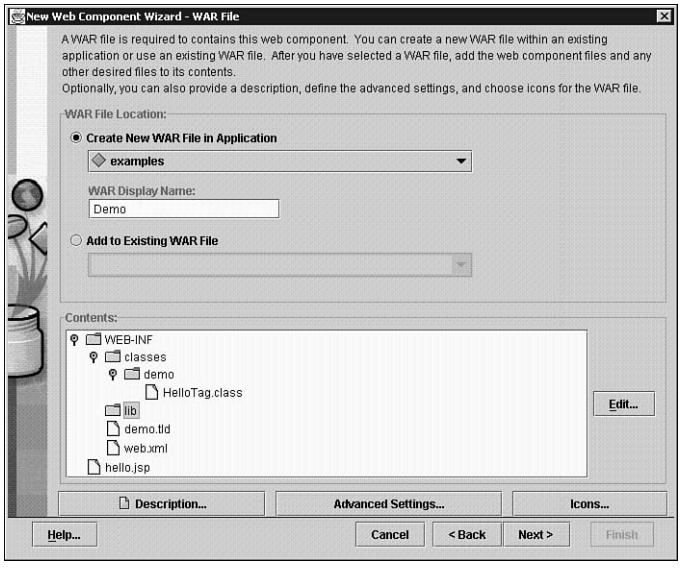
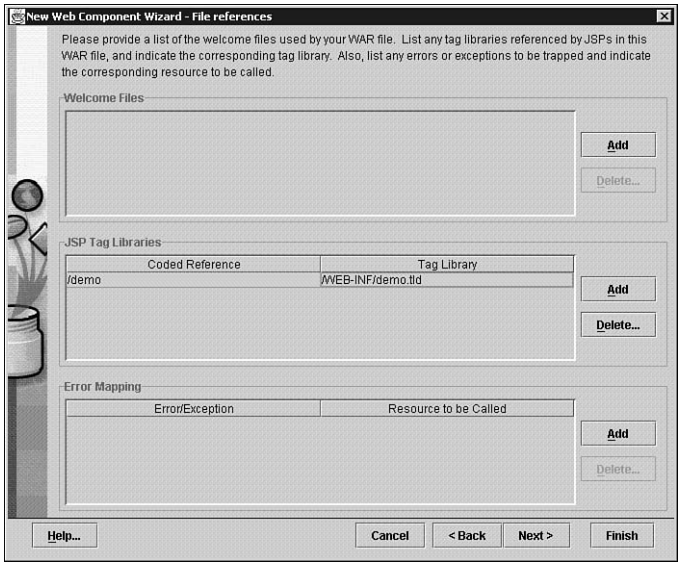

New Web Component Wizard - WAR File
A WAR file is required to contains this web component. You can create a new WAR file within an existing application or use an existing WAR file. After you have selected a WAR file, add the web component files and any other desired files to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.
WAR File Location:
Create New WAR File in Application
examples
WAR Display Name:
Demo
Add to Existing WAR File
Contents:
WEB-INF
classes
demo
HelloTag.class
lib
demo.tld
web.xml
hello.jsp
Edit...
Description...
Advanced Settings...
Icons...
Help...
Cancel
< Back
Next >
Finish

FIGURE 14.4

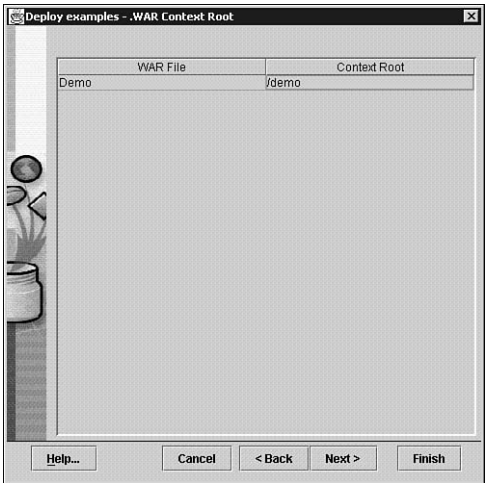
Defining the Tag Library TLD file.


New Web Component Wizard - File references
Please provide a list of the welcome files used by your WAR file. List any tag libraries referenced by JSPs in this WAR file, and indicate the corresponding tag library. Also, list any errors or exceptions to be trapped and indicate the corresponding resource to be called.
Welcome Files
Add
Delete...
JSP Tag Libraries
Coded Reference
Tag Library
/demo
/WEB-INF/demo.tld
Add
Delete...
Error Mapping
Error/Exception
Resource to be Called
Add
Delete...
Help...
Cancel
< Back
Next >
Finish

10. Now run the `verifier` on your Web Application to check for any errors and inconsistencies in your JSP, TLD, and class files. You should not find any errors if you are using the example code provided on the CD-ROM that comes with this book.

11. Your application can now be deployed using the Tools, Deploy menu option.
12. On the first deploy screen, simply click Next (you do not need to return the client JAR file).
13. On the next Root Context screen, define a context directory for your application. The example shown in Figure 14.5 has set the context to `/demo`.

FIGURE 14.5
Defining the Web context.



14. Use your browser to access the URL `http://localhost:8000/demo/hello` and the Hello World Web page will be displayed.

Defining the TLD Location

The example JSP page in Listing 14.2 used a logical name to access the TLD. The logical name must be defined in the Web Application Deployment Descriptor (`web.xml`). This is considered the best approach, because the JSP is not dependent on the physical location of the TLD.

An alternative approach is to use the actual TLD file location as shown in the following:

```
<%@ taglib uri="/WEB-INF/demo.tld" prefix="demo" %>
```

The location of the TLD is defined relative to the location of the Web Application.

Using Simple Tags

You would not normally write tags as simple as the `Hello World` example. Even so, simple tags like this do serve a purpose. They can be used to retrieve information from the

system and display it via the JSP page. The mechanics of accessing the information are hidden inside the custom tag code and are not shown on the JSP page. Hiding complex operations (perhaps a JNDI lookup operation or obtaining data from a database) helps simplify the writing of the Web page.

However, without the ability to pass information into the tag, this functionality has limited use.

Tag Libraries support the following two ways of passing information from the page into the tag:

- Via parameters passed in with the tag
- Via the text in the body of the tag

Using parameters is the most common approach and is discussed in detail below. Use of tag-specific body data is discussed in the "Processing Tag Bodies" section near the end of today's work.

Tags with Attributes

Attributes are used to pass information into a custom tag to configure the behavior of the tag. Tag attributes are like XML or HTML attributes and are name/value pairs of data. The values must be quoted with single or double quotes.

A simple example is a tag that looks up a JNDI name passed in as a parameter. Listing 14.4 shows how such a tag could be used.

LISTING 14.4 Full Text of `lookup.jsp`

```
1: <%@ taglib uri="/demo" prefix="demo" %>
2: <HTML>
3: <HEAD>
4: <TITLE>Tag Library Lookup Demo</TITLE>
5: <BODY>
6:   <demo:lookup name="jdbc/Agency"/>
7: </BODY>
8: </HTML>
```

The JSP specification allows attributes to have values that can include Java expressions. If the lookup name had been passed in as a request parameter called `agencyJNDIname`, the lookup tag could have been written as follows:

```
<demo:lookup name="<%=request.getParameter("agencyJNDIname")%>"/>
```

Java expressions like this are referred to as *request time expressions*. Tag attributes can be defined to disallow request time expressions, in which case, the value must be a simple string (as shown in Listing 14.4).

The TLD description for the tag must define any attributes it uses. Each supported attribute name must be listed together with details of the attribute. Every attribute must have an `<attribute>` tag with the sub-components shown in Table 14.4.

TABLE 14.4 TLD Tags for Defining Attributes

Tag	*Description*
`attribute`	Introduces an attribute definition in the `<tag>` component of the TLD.
`name`	Defines the attribute name.
`required`	Followed by `true` if the attribute must be provided; otherwise, `false`.
`rtexprvalue`	Defines whether the attribute can be specified with a request time expression. Set this element to `true` to allow JSP scripting values to be used for the attribute; otherwise, the value must be a string literal.
`type`	Defines the type of an attribute and defaults to `java.lang.String`. This element must be defined if the `rtexprvalue` is `true` and the attribute value is not a String.

Listing 14.5 shows the complete TLD entry for the `lookup` tag.

LISTING 14.5 TLD Entry for the `lookup` Tag

```
 1: <tag>
 2:     <name>lookup</name>
 3:     <tag-class>demo.LookupTag</tag-class>
 4:     <body-content>empty</body-content>
 5:     <attribute>
 6:       <name>name</name>
 7:       <required>true</required>
 8:       <rtexprvalue>true</rtexprvalue>
 9:     </attribute>
10:   </tag>
```

For a tag to support attributes, it must follow the JavaBean idiom of providing get and set methods for manipulating the attributes as shown in Listing 14.6, which shows the Java implementation of the `lookup` tag.

LISTING 14.6 Full Text of `LookupTag.java`

```
1: package demo;
2:
```

LISTING 14.6 Continued

```
 3: import javax.naming.*;
 4: import javax.servlet.jsp.*;
 5: import javax.servlet.jsp.tagext.*;
 6:
 7: public class LookupTag extends TagSupport {
 8:     private String name;
 9:
10:     public String getName() {
11:         return name;
12:     }
13:
14:     public void setName(String name) {
15:         this.name = name;
16:     }
17:
18:     public int doStartTag() throws JspException {
19:         String msg = null;
20:         try {
21:             Context ic = new InitialContext();
22:             msg = ic.lookup(name).toString();
23:         }
24:         catch (NamingException ex) {
25:              msg = ex.toString();
26:            }
27:         catch (ClassCastException ex) {
28:              msg = ex.toString();
29:         }
30:         try {
31:             pageContext.getOut().print(msg);
32:         }
33:         catch (Exception ex) {
34:             throw new JspTagException("LookupTag: "+ex);
35:         }
36:         return SKIP_BODY;
37:      }
38:
39:     public int doEndTag() {
40:         return EVAL_PAGE;
41:     }
42: }
```

The set method for each attribute specified for the tag is called prior to the `doStartTag()` method. The `doStartTag()` and other tag lifecycle methods can use the values of attributes to determine their behavior.

Tags that Define Script Variables

Now that you know the basics of custom tags, you can start developing more functional tags for use in your applications. The first feature you will study is that of creating scripting variables that can be used on the Web page.

A scripting variable can be added to the page context using the `setAttribute()` method of the instance variable called `pageContext` that points to the `javax.servlet.jsp.PageContext` object of the Web page. The `setAttribute()` method takes the following parameters:

- The name of the scripting variable. This name is used to refer to the variable on the Web page.
- The object reference to the variable to be added to the page.
- The scope of the variable that can be one of the following constants defined in `PageContext`:
  - `APPLICATION_SCOPE` Available until the context is reclaimed
  - `PAGE_SCOPE` Available until the current page processing completes (this is the default)
  - `REQUEST_SCOPE` Available until the current request completes allowing the variable to be used by other pages should this page forward the HTTP request
  - `SESSION_SCOPE` Available to all pages in the current session

The following example adds a `String` variable to the context under the name `"title"` and available to this page and all forwarded pages in this request:

```
String s = "Example Title";
pageContext.setAttribute("title", s, PageContext.REQUEST_SCOPE);
```

To use scripting variables, you will also need to define each scripting variable in the TLD file. Every scripting variable must have a `<variable>` tag with the sub-components shown in Table 14.5.

TABLE 14.5 TLD Tags for Defining Variables

Tag	*Description*
`name-given`	Defines the name for the scripting variable as a fixed value (cannot be specified if `name-from-Attribute` is defined)
`name-from-attribute`	Specifies the attribute that is used to define the scripting variable name (cannot be specified if `name-given` is defined)

TABLE 14.5 Continued

Tag	*Description*
`variable-class`	Specifies the class of the scripting variable
`declare`	Specifies if the variable is a new object (defaults to `true`)
`scope`	Defines the scope of the variable—must be one of `NESTED`, `AT_BEGIN`, or `AT_END` and refer to where in the Web page the variable can be accessed

Using the example `String` variable would require the following entry in the `<tag>` section of the TLD:

```
<variable>
  <name-given>title</name-given>
  <variable-class>java.lang.String</variable-class>
  <declare>true</declare>
  <scope>AT_BEGIN</scope>
</variable>
```

Variable components must be defined before any attribute components for the tag.

You can use scripting variables in the Job Agency case study pages you developed on Day 13, "JavaServer Pages," to remove one of the perceived weaknesses of defining beans that require initialization parameters.

As a first step, consider the following code fragment from the `advertise.jsp`:

```
<jsp:useBean id="cust" class="web.CustomerBean" scope="request" >
  <jsp:setProperty name="cust" property="login" param="customer"/>
</jsp:useBean>
```

This fragment creates a JavaBean for the customer information. This code looks clumsy. Ideally, you should be able to create the bean and pass in the initialization properties in a single tag. The JSP idiom of setting the bean properties inside the `useBean` start and end tags is a contrived solution forced on you by the way beans are used in the JSP. With custom tags, you can provide a much cleaner solution. The following `getCust` tag creates the required bean and adds it to the page context:

```
<agency:getCust login='<%=request.getParameter("customer")%>'/>
```

Looking back at the bean (`CustomerBean.java`) used to access the customer details, you will see it is a simple adapter for the `Advertise` Session bean. Its only purpose is to adapt the Session bean interface to the JavaBean semantics required on the Web page. The following code fragment shows the key points of the `Bean` class:

```
public CustomerBean () throws NamingException {
    InitialContext ic = new InitialContext();
    advertiseHome = (AdvertiseHome)ic.lookup("java:comp/env/ejb/Advertise");
```

```
}
public void setLogin (String login) throws Exception {
    this.login = login;
    advertise = advertiseHome.create(login);
}
public String getName() throws RemoteException    {
    return advertise.getName();
}
```

- The bean constructor obtains the home object for the Session bean.
- The `setLogin()` method creates the bean as a side effect of setting the login attribute.
- All the other methods (such as `getName()`) simply delegate to the appropriate method in the Session bean.

The advertise Session EJB was developed to conform to the JavaBean semantics for accessing and setting attributes. Using a custom tag, you can use the Session EJB directly without the need for a bean wrapper class. Listing 14.7 shows the customer tag that creates the `Advertise` Session bean for the `advertise.jsp` page.

LISTING 14.7 Full Text of `GetCustTag.java`

```
 1: package web;
 2:
 3: import javax.naming.*;
 4: import javax.servlet.jsp.*;
 5: import javax.servlet.jsp.tagext.*;
 6: import agency.*;
 7:
 8: public class GetCustTag extends TagSupport {
 9:     private String login;
10:
11:     public String getLogin() {
12:         return login;
13:     }
14:
15:     public void setLogin(String login) {
16:         this.login = login;
17:     }
18:
19:     public int doStartTag() throws JspException {
20:         try {
21:            InitialContext ic = new InitialContext();
22:            AdvertiseHome advertiseHome =
➥ (AdvertiseHome)ic.lookup("java:comp/env/ejb/Advertise");
23:            Advertise advertise = advertiseHome.create(login);
24:            pageContext.setAttribute("cust",
➥advertise, PageContext.REQUEST_SCOPE);
```

LISTING 14.7 Continued

```
25:         }
26:         catch (Exception ex) {
27:             throw new JspTagException("CustomerTag: "+ex);
28:         }
29:         return SKIP_BODY;
30:      }
31:
32:     public int doEndTag() {
33:         return EVAL_PAGE;
34:     }
35: }
```

The `doStartTag()` method finds and creates the `Advertise` Session EJB using the login name passed as an attribute to the tag. The created bean is added to the page context (line 24) using the name `cust`, and its scope is set to the current request. After the `getCust` tag has been defined, the rest of the page can refer to the `cust` bean and use the `jsp:getProperty` and `jsp:setProperty` tags. Because the session bean uses the same properties as the `CustomerBean` wrapper class, there are no additional changes required to the JSP code.

Listing 14.8 shows the `getCust` TLD entry with the `variable` and `attribute` tags. Note that the `login` attribute must have an `rtexprvalue` of `true` to allow the Web page to pass in the value from the HTTP request parameter.

LISTING 14.8 `getCust` Tag Entry in `agency.tld`

```
 1: <tag>
 2:     <name>getCust</name>
 3:     <tag-class>web.GetCustTag</tag-class>
 4:     <body-content>empty</body-content>
 5:     <variable>
 6:       <name-given>cust</name-given>
 7:       <variable-class>agency.Advertise</variable-class>
 8:       <declare>true</declare>
 9:       <scope>AT_BEGIN</scope>
10:     </variable>
11:     <attribute>
12:       <name>login</name>
13:       <required>true</required>
14:       <rtexprvalue>true</rtexprvalue>
15:     </attribute>
16: </tag>
```

To keep the example code simple, it always stores the bean against the name `cust`. To be more flexible, you could add an additional parameter (such as `beanName`) to be used to allow the customer bean variable name to specified by the JSP developer. In this case, you would set the TLD `<variable>` information to use this parameter using the `<name-from-attribute>beanName<name-from-attribute/>` tag.

You can update the other Web pages in the case study that use the `CustomerBean` class in a similar manner, and develop a `GetJobTag.java` to remove the need for the `JobBean` wrapper around the `AdvertiseJob` Session bean. The worked case study example includes these updates; later, your task, should you choose to accept it, will be to update the applicant registration pages to use custom tags.

The new versions of the Web pages can be deployed now, or you can wait and learn more about custom tags to address the restrictions of JSP when processing repetitive data.

Iterative Tags

One of the problems with processing dynamic data is that HTML and JSP tags do not support repetitive data very well. There is no way of defining the layout of one row of data and asking for this to be applied to all subsequent rows; each row has to be defined explicitly in the page.

Iterative custom tags interact with the processing of the start and end tags to ask for the tag body to be processed again, and again, and again….

An iterative tag must implement the `IterationTag` interface. This is most commonly achieved by sub-classing the `BodyTagSupport` class. The `doAfterBody()` method must return `IterationTag.EVAL_BODY_AGAIN` to process the body again or `Tag.SKIP_BODY` to stop the iteration. Typically, the `doAfterBody()` method will change the data for each iteration loop.

The iteration tag has complete control over the body content of the page because it can return values from the page interaction methods, such as `doAfterBody()` and then tell the JSP processor how to continue processing the page.

The default behavior for the `BodyTagSupport` class is to buffer up the body text of the custom tag and discard it when the end tag is processed. You will have to override this behavior in your custom tag so that it outputs the body text either every time round the iteration loop or once at the end of the tag.

The `BodyTagSupport` class stores the body content in a `BodyTagSupport` class instance variable called `bodyContent` (class `javax.servlet.jsp.tagext.BodyContent`). The `BodyContent` class extends the `JSPWriter` class.

The following code illustrates the normal approach to writing the buffered body content to the Web page:

```
JspWriter out = getPreviousOut();
out.print(bodyContent.getString());
bodyContent.clearBody();
```

The steps required are as follows:

1. Obtain the `JSPWriter` object that can be used to output the body content text (the `getPreviousOut()` method).
2. Print the string data buffered up in the body content.
3. Clear the body content text after it has been written to the page; otherwise, it will be written more than once.

Typically, the body content is added to the page from within the `doAfterBody()` method. This keeps the size of the body content down because it is flushed to the page with each iteration and saves on memory usage. If the body content cannot be determined until all of the iterations have completed (or the tag is not an iterative one), you will have to write it to the page in the `doEndTag()` method.

Listing 14.9 revisits the case study and shows a custom tag for supporting iteration over a customer's jobs.

LISTING 14.9 Full Text of `ForEachJobTag.java`

```
 1: package web;
 2:
 3: import java.rmi.*;
 4: import javax.ejb.*;
 5: import javax.naming.*;
 6: import javax.servlet.jsp.*;
 7: import javax.servlet.jsp.tagext.*;
 8: import agency.*;
 9:
10: public class ForEachJobTag extends BodyTagSupport {
11:     private Advertise customer;
12:     AdvertiseJobHome advertiseJobHome;
13:     private String[] jobs;
14:     private int nextJob;
15:
16:     public Advertise getCustomer() {
17:         return customer;
18:     }
19:
20:     public void setCustomer(Advertise customer) {
21:         this.customer = customer;
```

LISTING 14.9 Continued

```
  22:     }
  23:
  24:     public int doStartTag() throws JspException {
  25:         try {
  26:            InitialContext ic = new InitialContext();
  27:            advertiseJobHome = (AdvertiseJobHome)ic.lookup(
➥"java:comp/env/ejb/AdvertiseJob");
  28:            jobs = customer.getJobs();
  29:            int nextJob = 0;
  30:            return getNextJob();
  31:         }
  32:         catch (Exception ex) {
  33:             throw new JspTagException("ForEachJobTag: "+ex);
  34:         }
  35:      }
  36:
  37:     public int doAfterBody() throws JspException {
  38:         try {
  39:             JspWriter out = getPreviousOut();
  40:             out.print(bodyContent.getString());
  41:             bodyContent.clearBody();
  42:             return getNextJob();
  43:         }
  44:         catch (Exception ex) {
  45:             throw new JspTagException("ForEachJobTag: "+ex);
  46:         }
  47:     }
  48:
  49:     private int getNextJob() throws RemoteException, CreateException
  50:     {
  51:        if (nextJob >= jobs.length)
  52:            return SKIP_BODY;
  53:        AdvertiseJob advertiseJob = advertiseJobHome.create(
➥jobs[nextJob++],customer.getLogin());
  54:        pageContext.setAttribute("job", advertiseJob,
PageContext.REQUEST_SCOPE);
  55:        return EVAL_BODY_AGAIN;
  56:     }
  57: }
```

In Listing 14.9, the `doStartTag()` and `doAfterBody()` methods both call the private helper method `getNextJob()`. It is this method that loads the information about the next job and returns `EVAL_BODY_AGAIN` while there is still job information available. The value `SKIP_BODY` is returned to stop the iteration loop when the data is exhausted.

As in the previous `GetCustTag` shown in Listing 14.7, the `ForEachJobTag` creates a scripting variable to hold the data for use within the body of the iteration tag. In this case, the variable is always called `job` and refers to an `AdvertiseJob` Session bean.

An iteration tag must process the tag body, so the TLD tag entry must define the `<body-content>` component as `JSP` to show that the body contains normal JSP text. The attribute passed into the `ForEachJob` tag to control the iteration is called `customer` and is a reference to an `Advertise` Session bean (which defines the customer details). The TLD entry for `ForEachJobTag` is shown in Listing 14.10.

LISTING 14.10 TLD Entry for the `ForEachJobTag` Tag

```
 1: <tag>
 2:     <name>forEachJob</name>
 3:     <tag-class>web.ForEachJobTag</tag-class>
 4:     <body-content>JSP</body-content>
 5:     <variable>
 6:       <name-given>job</name-given>
 7:       <variable-class>agency.AdvertiseJob</variable-class>
 8:       <declare>true</declare>
 9:       <scope>AT_BEGIN</scope>
10:     </variable>
11:     <attribute>
12:       <name>customer</name>
13:       <required>true</required>
14:       <rtexprvalue>true</rtexprvalue>
15:       <type>agency.Advertise</type>
16:     </attribute>
17: </tag>
```

With this new tag, you can update the `advertise.jsp` page to process the jobs without resorting to coding the iteration loop as a Java scriptlet. The following code shows the relevant part of the `advertise.jsp` page you developed on Day 13.

```
<H2>Jobs</H2>
<% String[] jobs = cust.getJobs(); %>
<% for (int i=0; i<jobs.length; i++) {%>
  <jsp:useBean id="job" class="web.JobBean" scope="request">
    <jsp:setProperty name="job" property="customer" param="customer"/>
  </jsp:useBean>
  <% job.setRef(jobs[i]); %>
  <H3><jsp:getProperty name="job" property="ref"/></H3>
  <FORM action=updateJob>
  …
  </FORM>
<% } %>
```

Here, initialization of the `JobBean` was awkward because a job has a compound key made up of the customer login name and the job reference. The initialization of the bean set the customer reference as a bean property using the `<jsp:setProperty>` tag and called the bean's `setRef()` method directly to reload the bean with the new job details.

You can replace this inelegant approach with much neater and cleanly encapsulated code as shown next:

```
<H2>Jobs</H2>
<agency:forEachJob customer="<%=cust%>">
  <H3><jsp:getProperty name="job" property="ref"/></H3>
  <FORM action=updateJob>
  ...
  </FORM>
</agency:forEachJob>
```

The `forEachJob` tag passes in a reference to the `Advertise` Session bean using the `customer` attribute. This attribute is initialized to the scripting variable you defined in the `getCust` tag in the previous section.

The last improvement you will make to this Web page is to replace the handling of the job skills with a set of cooperating tags, as discussed in the next section.

Co-operating Tags

Cooperating tags are those that share information in some way. JSP information can be shared using the following mechanisms:

- Scripting variables
- Hierarchical (or nested) tags

Using Shared Scripting Variables

One means of writing cooperating tags is to use scripting variables to pass information between the tags. A tag can create a scripting variable that can be retrieved by a another tag on the current page. Depending on the scope of the scripting variable, it can be passed on to other pages in the same request or to pages in subsequent requests. Variables defined by custom tags have the same scoping rules as other variables defined on the page. Scripting variables are a very flexible means of passing information between tags.

A tag can retrieve a scripting variable from the page context using the `getAttribute()` method. This method takes the name of the variable as its parameter. An overloaded version of the `getAttribute()` method takes a second parameter that defines the scope of the variable to retrieve.

The tag needs to know the class of the variable it is retrieving. The following example retrieves a `String` variable stored under the name `"title"`:

```
String s = (String)pageContext.getAttribute("title");
```

Hierarchical Tag Structures

An alternative means of passing information between tags is to use a parent/child (or hierarchical) relationship. The parent (outer tag) contains information that is accessed by a child (inner) tag. The parent tag is often an iterative tag, and the child is used to retrieve information for each iteration.

The advantage of this approach over scripting variables is that the information can only be used in the correct context. The scope of the information can be constrained to the Web page between the start and end tags of the parent tag.

Two static methods are provided in the `javax.servlet.jsp.tagext.TagSupport` class for finding a parent tag from within the child:

- `TagSupport.findAncestorWithClass(from, class)` This method searches through the tag hierarchy until it finds a tag with the same class as the second parameter. The first parameter defines the start point of the search and is typically the `this` object.
- `TagSupport.getParent()` This method finds the immediately enclosing parent tag.

With this information, you can now redesign the `advertise.jsp` to remove the rest of the Java scriptlets from the Job Agency Web pages as shown in the following examples.

On the `advertise.jsp` page, two separate pages (`location.jsp` and `skills.jsp`) handled the job location and skills. Using cooperating tags, you can remove all the messy handling of the HTML select tags. The code for handling the job skills select list currently looks like the following:

```
<% String[] skills = job.getSkills(); %>
<%@include file="skills.jsp"%>
```

The code creates a Java variable called `skills` that is used by the `skills.jsp` page shown in Listing 14.11.

LISTING 14.11 Full Text of `skills.jsp`

```
1: <SELECT name="skills" multiple size="6">
2: <%
3:   Iterator allSkills = agency.getSkills().iterator();
4:   while (allSkills.hasNext()) {
5:     String s = (String)allSkills.next();
6:     boolean found = false;
7:     for (int si=0; !found && si<skills.length; si++)
8:         found = s.equals(skills[si]);
9:     if (found)
```

LISTING 14.11 Continued

```
10:         out.print("<OPTION selected>");
11:     else
12:         out.print("<OPTION>");
13:     out.print(s);
14:   }
15: %>
16: </SELECT>
```

The use of the Java variable to communicate between the two pages allowed the `register.jsp` page to use the same `skills.jsp` code to generate the applicant's skill list. This is an error-prone approach as the `skills.jsp` page will not work unless this variable is defined before the `skills.jsp` page is included.

In other words the `skills.jsp` page is not self contained. You will be able to write a much more elegant solution by using cooperating tags.

You will need two custom tags—one to control the iteration over a list of option values and the other to define whether each option is selected or not. Listing 14.12 shows a generic iteration tag (called `ForEachTag`) that takes a collection of strings to iterate over.

LISTING 14.12 Full Text of `ForEachTag.java`

```
 1: package web;
 2:
 3: import java.io.*;
 4: import java.util.*;
 5: import javax.servlet.jsp.*;
 6: import javax.servlet.jsp.tagext.*;
 7:
 8: public class ForEachTag extends BodyTagSupport {
 9:     private Collection collection;
10:     private Iterator it;
11:     private String currentValue;
12:
13:     public void setCollection (Collection collection) {
14:         this.collection = collection;
15:     }
16:
17:     public Collection getCollection () {
18:         return collection;
19:     }
20:
21:     public int doStartTag() {
22:         it = collection.iterator();
23:         return getNext();
```

LISTING 14.12 Continued

```
24:      }
25:
26:     public int doAfterBody() throws JspTagException {
27:         try {
28:             JspWriter out = getPreviousOut();
29:             out.print(bodyContent.getString());
30:             bodyContent.clearBody();
31:             return getNext();
32:         }
33:         catch (IOException ex) {
34:             throw new JspTagException("ForEachTag: "+ex);
35:         }
36:     }
37:
38:     private int getNext()
39:     {
40:        if (it.hasNext()) {
41:            currentValue = (String)it.next();
42:            return EVAL_BODY_AGAIN;
43:        }
44:        return SKIP_BODY;
45:     }
46:
47:     public String getCurrentValue() {
48:        return currentValue;
49:     }
50: }
```

This is similar to the previous iteration example with the addition of an accessor method called `getCurrentValue()` to obtain the current iteration value.

The nested option tag (`OptionTag`), shown in Listing 14.13, looks up the tag hierarchy using `findAncestorWithClass()` to find the enclosing `ForEachTag` object. The inner tag then gets the current iteration value from the found ancestor object and compares it to a list of values passed as an attribute to the inner tag. If the current value is in the supplied list, the option is tagged as a selected option.

LISTING 14.13 Full Text of `OptionTag.java`

```
1: package web;
2:
3: import java.io.*;
4: import java.util.*;
5: import javax.servlet.jsp.*;
6: import javax.servlet.jsp.tagext.*;
```

LISTING 14.13 Continued

```
 7:
 8: public class OptionTag extends TagSupport {
 9:     private String[] selected = new String[0];
10:
11:     public void setSelected (String[] selected) {
12:         this.selected = selected;
13:     }
14:
15:     public String[] getSelected () {
16:         return selected;
17:     }
18:
19:     public int doStartTag() throws JspTagException {
20:         try {
21:             ForEachTag loop = (ForEachTag)findAncestorWithClass(
➥this, ForEachTag.class);
22:             String value = loop.getCurrentValue();
23:             for (int i=0; i<selected.length; i++) {
24:                 if (value.equals(selected[i])) {
25:                     pageContext.getOut().print(
➥"<OPTION selected>"+value);
26:                     return SKIP_BODY;
27:                 }
28:             }
29:             pageContext.getOut().print("<OPTION>"+value);
30:         }
31:         catch (IOException ex) {
32:             throw new JspTagException("OptionTag: "+ex);
33:         }
34:         return SKIP_BODY;
35:      }
36:
37: }
```

The `OptionTag` example is inherently an extension to the HTML `OPTION` tag, whereas the `ForEach` example simply iterates over a generic list. Both tags are eminently reusable on other JSP pages.

The TLD entries for these tags are shown in Listing 14.14.

LISTING 14.14 TLD entries for `ForEachTag` and `OptionTag`

```
1: <tag>
2:     <name>forEach</name>
3:     <tag-class>web.ForEachTag</tag-class>
4:     <body-content>JSP</body-content>
```

LISTING 14.14 Continued

```
 5:     <attribute>
 6:       <name>collection</name>
 7:       <required>true</required>
 8:       <rtexprvalue>true</rtexprvalue>
 9:       <type>java.util.Collection</type>
10:     </attribute>
11:   </tag>
12:   <tag>
13:     <name>option</name>
14:     <tag-class>web.OptionTag</tag-class>
15:     <body-content>empty</body-content>
16:     <attribute>
17:       <name>selected</name>
18:       <required>false</required>
19:       <rtexprvalue>true</rtexprvalue>
20:       <type>java.util.String[]</type>
21:     </attribute>
22: </tag>
```

Note how the selected attribute of the option tag is an array of `String` objects and is optional. The code in the tag implementation defaults this attribute to an empty array if it is not specified; effectively, none of the select options will be selected.

You can now complete the refactoring of the `advertise.jsp` page by using the cooperating tags. The new tag attributes are defined using Java expressions. The only complication is the need to convert the single location string into an array. The following code highlights how clean the new code for handling the location is.

The following fragment shows the code you developed on Day 13 to define the list of available locations:

```
<TR><TD>Location:</TD><TD>
  <% String location = job.getLocation(); %>
  <%@include file="location.jsf"%>
</TD></TR>
```

The included file `location.jsf` defines the `SELECT` list for the location as follows:

```
<SELECT name="location">
<%
  Iterator locations = agency.getLocations().iterator();
  while (locations.hasNext()) {
    String l = (String)locations.next();
    if (location == l)
      out.print("<OPTION>");
    else
```

```
      out.print("<OPTION selected>");
    out.print(l);
  }
%>
</SELECT>
```

You can replace this convoluted approach with a neater version using the two custom tags just shown:

```
<SELECT name="location">
  <agency:forEach collection='<%=agency.getLocations()%>'>
    <agency:option selected='<%=new String[]{job.getLocation()}%>'/>
  </agency:forEach>
</SELECT>
```

The complete version of the revised Advertise Job Web page is shown in Listing 14.15.

LISTING 14.15 Full Text of `advertise.jsp`

```
 1: <%@ taglib uri="/agency" prefix="agency" %>
 2: <HTML><HEAD>
 3: <TITLE>Advertise Customer Details</TITLE>
 4: <%@include file="header.jsp" %>
 5: <%@page import="java.util.*" %>
 6: <agency:getCust login='<%=request.getParameter("customer")%>'/>
 7: <H2>Customer details for:
➥<jsp:getProperty name="cust" property="login"/></H2>
 8: <FORM action=updateCustomer>
 9: <INPUT type=hidden name=login
➥value="<jsp:getProperty name="cust" property="login"/>">
10: <TABLE>
11:   <TR><TD>Login:</TD><TD><jsp:getProperty
➥name="cust" property="login"/></TD></TR>
12:   <TR><TD>Name:</TD><TD><input type=text name=name value=
➥"<jsp:getProperty name="cust" property="name"/>"></TD></TR>
13:   <TR><TD>Email:</TD><TD><input type=text name=email value=
➥"<jsp:getProperty name="cust" property="email"/>"></TD></TR>
14:   <% String[] address = cust.getAddress(); %>
15:   <TR><TD>Address:</TD><TD>
➥<input type=text name=address value="<%=address[0]%>"></TD></TR>
16:   <TR><TD>Address:</TD>
➥<TD><input type=text name=address value="<%=address[1]%>"></TD></TR>
17: </TABLE>
18: <INPUT type=submit value="Update Details">
19: <INPUT type=reset>
20: </FORM>
21: <FORM action=deleteCustomer>
22: <input type=hidden name=customer value="<jsp:getProperty name=
➥"cust" property="login"/>">
23: <input type=submit value="Delete Customer <jsp:getProperty name=
```

LISTING 14.15 Continued

```
➥"cust" property="login"/>">
  24: </FORM>
  25: <H2>Jobs</H2>
  26: <agency:forEachJob customer="<%=cust%>">
  27:   <H3><jsp:getProperty name="job" property="ref"/></H3>
  28:   <FORM action=updateJob>
  29:     <TABLE>
  30:       <TR><TD>Description:</TD><TD>
➥<input type=text name=description value="<jsp:getProperty
➥name="job" property="description"/>"></TD></TR>
  31:       <TR><TD>Location:</TD>
  32:         <TD>
  33:         <SELECT name="location">
  34:           <agency:forEach collection='<%=agency.getLocations()%>'>
  35:             <agency:option
➥selected='<%=new String[]{job.getLocation()}%>'/>
  36:           </agency:forEach>
  37:         </SELECT>
  38:         </TD>
  39:       </TR>
  40:       <TR><TD>Skills:</TD><TD>
  41:       <% String[] skills = job.getSkills(); %>
  42:       <SELECT name=skills multiple size=6>
  43:       <agency:forEach collection='<%=agency.getSkills()%>'>
  44:         <agency:option selected='<%=job.getSkills()%>'/>
  45:       </agency:forEach>
  46:       </SELECT>
  47:       </TD></TR>
  48:     </TABLE>
  49:     <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="job" property="customer"/>">
  50:     <INPUT type=hidden name="ref"
➥value="<jsp:getProperty name="job" property="ref"/>">
  51:     <INPUT type=submit value="Update Job">
  52:   </FORM>
  52:   <FORM action=deleteJob>
  54:     <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="job" property="customer"/>">
  55:     <INPUT type=hidden name="ref"
➥value="<jsp:getProperty name="job" property="ref"/>">
  56:     <INPUT type="submit"
➥value='Delete Job <jsp:getProperty name="job" property="ref"/>'>
  57:   </FORM>
  58: </agency:forEachJob>
  59: <H2>Create New Job</H2>
  60: <FORM action=createJob>
  61: <TABLE>
  62: <TR>
```

LISTING 14.15 Continued

```
63:   <TD>Ref:</TD>
64:   <TD><INPUT type=text name=ref></TD>
65: </TR>
66: <TR>
67:   <INPUT type=hidden name="customer"
➥value="<jsp:getProperty name="cust" property="login"/>">
68:   <TD colspan=2><INPUT type=submit value="Create Job"></TD>
69: </TR>
70: </TABLE>
71: </FORM>
72: <%@include file="footer.jsp" %>
73: </BODY>
74: </HTML>
```

The `ForEach` tag was designed to be useful for any kind of iteration of values to show you how generic tags can be developed. In this example, it may have been better to have an iteration tag that implemented the HTML `SELECT` list rather than require the developer to define them on the JSP page. Such a custom tag would need to have attributes matching the attributes of the HTML `SELECT` tag. The JSP code using a custom `<select>` tag could look like the following:

```
<agency:select name="skills" multiple="true" size="6"
➥collection="<%=agency.getSkills()%>">
   <agency:option selected="<%=job.getSkills()%>"/>
</agency:select>
```

One of the problems with more complex custom tags is that there are often restrictions on how the attributes can be defined that can't be shown in TLD definition. A tag may require that an attribute have specific values (the prevous `size` tag must be a positive integer) or perhaps two tags are mutually exclusive. Custom tags address this problem by using a Tag Extra Info (TEI) object, as discussed in the next section.

Defining Tag Extra Info Objects

Every custom tag can have an optional Tag Extra Info (TEI) object. The TEI object is used for two purposes:

- To validate tag attributes at page translation time
- To specify the scripting variables created by the tag

A TEI class extends the `javax.servlet.jsp.tagext.TagExtraInfo` class and is defined to the TLD before the tag's body content definition. The following example defines an `OptionTagTEI` class for the `OptionTag` used in the case study:

```
<tag>
    <name>option</name>
    <tag-class>web.OptionTag</tag-class>
    <tei-class>web.OptionTagTEI</tei-class>
    <body-content>empty</body-content>
```

The TEI class itself must define methods to validate attributes and define scripting variables. These uses of the TEI class are presented in the next two sub-sections.

Validating Attributes

As you know from Day 13, when a JSP is first accessed, the JSP is translated into a Java servlet that is then compiled. After the compiled servlet is available, it is used to process the client's HTTP request. The process of translating the JSP text and compiling the servlet can fail (and frequently does during page development). As explained on Day 13, some of the most common errors are as follows:

- JSP syntax errors
- Compilation errors in embedded Java scriptlets
- Incorrectly specifying tag attributes

The first of these errors can be reduced by using JSP, HTML, and XML syntax-aware editors.

The second problem is reduced when custom tags are used to encapsulate Java code, enabling Java compilation errors to be detected and corrected as the tag is developed.

Solving the third problem of misusing attributes for a custom tag is addressed by defining a Tag Extra Info class to validate your attributes. You would do this if you had specific rules about which attributes could be defined or the value an attribute could have.

As an example, you can update the `Option` tag from the previous section to allow a second attribute, called `default`, which could be used when a single option is required rather than force the developer to construct an array of strings. The `selected` and `default` attributes for your tag would be mutually exclusive. The TLD entry defining the `<variable>` tag cannot express this constraint, but you can by using a TEI class.

The TEI class defines a single method called `isValid()`. The single parameter is a `javax.servlet.jsp.tagext.TagData` that provides a `getAttribute()` method to access each of the tag's attributes. The example in Listing 14.16 shows how the revised `Option` tag could validate that the `default` and `selected` attributes are not both defined.

LISTING 14.16 Full Text of `OptionTagTEI.java`

```
 1: package web;
 2:
 3: import javax.servlet.jsp.*;
 4: import javax.servlet.jsp.tagext.*;
 5:
 6: public class OptionTagTEI extends TagExtraInfo {
 7:    public boolean isValid(TagData data) {
 8:       Object selected = data.getAttribute("selected");
 9:       Object def = data.getAttribute("default");
10:       if (def!=null && selected!=null)
11:           return false;
12:       else
13:           return true;
14:    }
15: }
```

The `advertise.jsp` can be updated to use the default attribute for accessing the location list:

```
<agency:option default='<%=job.getLocation()%>'/>
```

The `OptionTag.java` class needs updating to support the `default` attribute. This is a simple change that adds the `getDefault()` and `setDefault()` bean methods shown next:

```
public void setDefault (String selected) {
        this.selected = new String[]{selected};
    }

    public String getDefault () {
        return selected[0];
    }
```

To use the Tag Extra Info to validate the value of a parameter, you will need to verify that the value is not a request time expression. Only if the value is a static Java `String` can it be compared against the permitted values. The example in Listing 14.17 verifies that a tag's color attribute can only take the values red, yellow, amber and green.

LISTING 14.17 TEI Example to Validate a Color Attribute

```
1: public class SignalTagTEI extends TagExtraInfo {
2:    public boolean isValid(Tagdata data) {
3:       Object o = data.getAttribute("color");
4:       if (o != null && o != TagData.REQUEST_TIME_VALUE) {
5:           String color = (String)o;
6:           if (col.equals("red") || col.equals("yellow") ||
7:              col.equals("amber") || col.equals("green"))
```

LISTING 14.17 Continued

```
 8:             return true;
 9:           else
10:             return false;
11:       }
12:       else
13:          return true;
14:    }
15: }
```

Defining Scripting Variables

The second use of the Tag Extra Info class is to provide information about the scripting variables used by the tag. This TEI information is not required for deploying a tag that creates scripting variables, but it does provide information that can be used by some manufacturer's JSP design tools.

The following example shows how the `getCust` tag shown in Listing 14.7 could define the `cust` scripting variable:

```
public class DefineTei extends TagExtraInfo {
   public VariableInfo[] getVariableInfo(TagData data) {
      String type = data.getAttributeString("type");
      if (type == null)
         type = "java.lang.Object";
      return new VariableInfo[] {
               new VariableInfo(data.getAttributeString("id"),
              type, true, VariableInfo.AT_BEGIN)};
   }
}
```

The name (`cust`) and class (`agency.Advertise`) of the scripting variable are passed in as tag attributes. The `TagData.getAttributeString()` method can retrieve this information and use it to construct the `VariableInfo` object required as a return value from this method. The scope of the variable is set to be `VariableInfo.AT_BEGIN` to allow it to be used after the opening tag.

Processing Tag Bodies

So far, you have seen tags that treat the text in the body of the tag as simple JSP (or HTML) data. But tags can also manipulate the tag body in any way they choose. A simple example would be a tag that interprets its body as an SQL statement and uses this to query the database. Listing 14.18 shows a simple use of such a tag.

LISTING 14.18 Full Text of `select.jsp`

```
 1: <%@ taglib uri="/demo" prefix="demo" %>
 2: <HTML>
 3: <HEAD>
 4: <TITLE> Tag Library SQL Query Demo</TITLE>
 5: <BODY>
 6:   <demo:select>
 7:     SELECT * FROM Customer
 8:   </demo:select>
 9: </BODY>
10: </HTML>
```

The implementation of this `select` tag must state in the TLD that the body text is tag dependent, as shown in the following:

```
<tag>
    <name>select</name>
    <tag-class>demo.SelectTag</tag-class>
    <body-content>tagdependent</body-content>
</tag>
```

The implementation of the `select` tag must extend the `BodyTagSupport` class as shown in Listing 14.19. This tag tags an SQL select statement as the tag body and runs this statement against a database (in this case, the Agency Case study database).

LISTING 14.19 Full Text of `SelectTag.java`

```
 1: package demo;
 2:
 3: import java.io.*;
 4: import java.sql.*;
 5: import javax.sql.*;
 6: import javax.naming.*;
 7: import javax.servlet.jsp.*;
 8: import javax.servlet.jsp.tagext.*;
 9:
10: public class SelectTag extends BodyTagSupport {
11:    public int doEndTag() throws JspException {
12:       try {
13:          String ans = doSelect(bodyContent.getString());
14:          pageContext.getOut().print(ans);
15:          bodyContent.clearBody();
16:       }
17:       catch (Exception ex) {
18:          throw new JspTagException("SqlTag: "+ex);
19:       }
20:       return EVAL_PAGE;
21:    }
22:
23:    private String doSelect (String cmd)
```

LISTING 14.19 Continued

```
➥throws NamingException, SQLException {
  24:       StringBuffer ans = new StringBuffer("<H2>"+cmd+"</H2>");
  25:       InitialContext ic = new InitialContext();
  26:       DataSource dataSource =
➥(DataSource)ic.lookup("java:comp/env/jdbc/Agency");
  27:       Connection con = null;
  28:       PreparedStatement stmt = null;
  29:       ResultSet rs = null;
  30:       try {
  31:           con = dataSource.getConnection();
  32:           stmt = con.prepareStatement(cmd);
  33:           rs = stmt.executeQuery();
  34:           ans.append("<TABLE border=1>");
  35:           ResultSetMetaData rsmd = rs.getMetaData();
  36:           int numCols = rsmd.getColumnCount();
  37:           // get column header info
  38:           ans.append("<TR>");
  39:           for (int i=1; i <= numCols; i++) {
  40:               ans.append("<TH>");
  41:               ans.append(rsmd.getColumnLabel(i));
  42:               ans.append("</TH>");
  43:           }
  44:           ans.append("</TR>");
  45:           // get cmma separated data
  46:           while (rs.next()) {
  47:               ans.append("</TR>");
  48:               for (int i=1; i <= numCols; i++) {
  49:                   ans.append("<TD>");
  50:                       ans.append(rs.getString(i));
  51:                   ans.append("</TD>");
  52:               }
  53:               ans.append("</TR>");
  54:           }
  55:           ans.append("</TABLE>");
  56:       }
  57:       finally {
  58:           if (rs != null) {
  59:               try {rs.close();} catch (SQLException ex) {}
  60:           }
  61:           if (stmt != null) {
  62:               try {stmt.close();} catch (SQLException ex) {}
  63:           }
  64:           if (con != null) {
  65:               try {con.close();} catch (SQLException ex) {}
  66:           }
  67:       }
  68:       return ans.toString();
  69:    }
  70: }
```

The `BodyTagSupport` processing buffers up the body text in the `bodyContent` variable and the `doEndTag()` method retrieves the tag body and hands it on to the private helper method `doSelect()` for execution as an `SQL SELECT` statement.

The `doSelect()` method queries the database (using a registered JNDI data source) and formats the results as an HTML table. The `doAfterBody()` method prints out the results of the SQL query and clears down the body content buffer.

Listing 14.19 shows a very simple example of what can be achieved by using custom tags. A simple variation of this Web page is to provide a form that lets the user select the customer name and change the query to show the details for a single customer passed in as a request parameter called `"customer"`. The new JSP would be like the following:

```
<%@ taglib uri="/demo" prefix="demo" %>
<HTML>
<HEAD>
<TITLE> Tag Library SQL Query Demo</TITLE>
<BODY>
  <demo:select>
    SELECT * FROM Customer WHERE login = <%=request.getParamter("customer")%>
  </demo:select>
</BODY>
</HTML>
```

The possibilities of this approach are endless. The fact that any custom tag can retrieve the body content and manipulate the text provides complete flexibility in how a JSP can be written.

JavaServer Pages Standard Tag Library (JSPTL)

Tag Libraries are a recent addition to JSP and have been quickly adopted by the development community. As with all new technologies, Tag Libraries are rapidly developing and changing.

As you saw in the section on cooperating tags, it is easy to write generic tags that provide useful functionality, such as iteration. Many Tag Libraries provided with JSP-compliant Web servers include their own custom tags, many of which are generic tags.

The proliferation of different Tag Libraries providing similar features has been seen as a problem, and the Java Community Process (JCP) is working on a standard set of tags to be included in the JSP standard.

The JavaServer Pages Standard Tag Library (JSPTL) is Java Community Process initiative and is specified on the JCP Web page at `http://jcp.org/jsr/detail/052.jsp`.

The latest implementation of the JSPTL can be downloaded from the Apache Jakarta TagLibs page at
`http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html`.

The Jakarta project also provides a large number of other tag libraries addressing common page requirements, such as support for JNDI, JMS, and JDBC.

Using the JSPTL with the J2EE RI

Using the JSPTL libraries with the J2EE is a simple process. You start by downloading the JSPTL archive from the Apache Jakarta Web site at
`http://jakarta.apache.org/builds/jakarta-taglibs/releases/standard/`.

Extract the downloaded archive and in the JSPTL directory, you will see a number of files, including two TLD files and some JAR files.

JSPTL defines two versions of the Tag Libraries. One version uses standard Java request time expressions, and the other uses a suggested scripting language. The two supplied TLD files refer to the two libraries known as `jr` (runtime expressions in file `jsptl-tr.tld`) and `jx` (scripting language in file `jsptl-jx.tld`).

In this section, you will use the JSPTL `jr` library to examine some of the features of the standard tags. You will need to include the JSPTL JAR file and TLD in your application. Look in the JSPTL directory for the `jsptl.jar` file (it may have a version number, such as `jsptil-1.2.jar`, or it may be an early access version `jsptlea.jar`).

You are now ready to include the JSPTL in your Web application. Start up `deploytool` and, to keep the demonstration simple, create a new application called `jsptl`. Create a new Web application and add the JSPTL JAR file (`jsptl.jar`) and the `jr` TLD file (`jsptl-tr.tld`). Figure 14.6 shows the file list from the Web Application Wizard (the JAR file has been added to the `WEB-INF/lib` directory).

Click on the Next button and on the Component Type page select the No Component option. Click on Next six times to get to the File Refs page and then add an entry for the TLD file. Figure 14.7 shows the entry mapping the logical name `/jsptl-jr` onto the actual file `/WEB-INF/jsptl-jr.tld`.

FIGURE 14.6

Adding the JSPTL files.

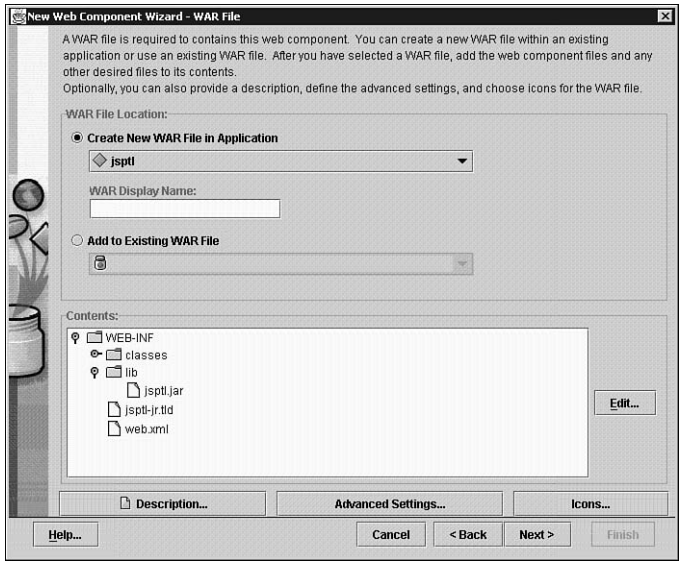
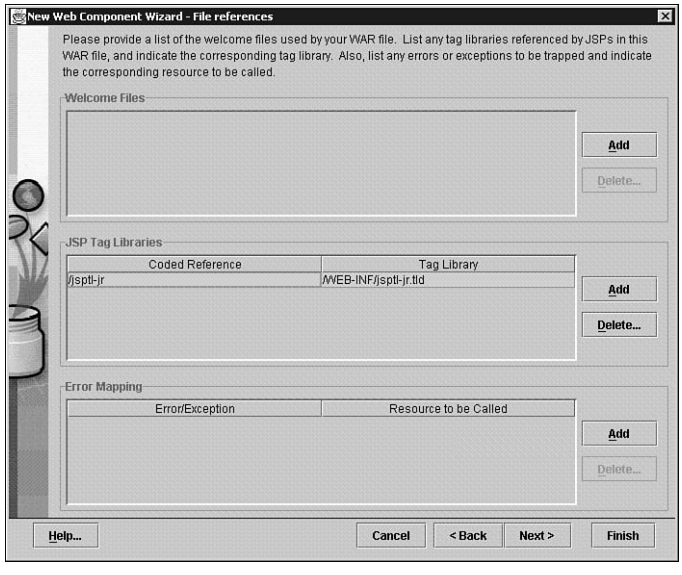

New Web Component Wizard - WAR File
A WAR file is required to contains this web component. You can create a new WAR file within an existing application or use an existing WAR file. After you have selected a WAR file, add the web component files and any other desired files to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.
WAR File Location:
Create New WAR File in Application
jsptl
WAR Display Name:
Add to Existing WAR File
Contents:
WEB-INF
classes
lib
jsptl.jar
jsptl-jr.tld
web.xml
Edit...
Description...
Advanced Settings...
Icons...
Help...
Cancel
< Back
Next >
Finish

FIGURE 14.7

Defining the JSPTL TLD file reference.


New Web Component Wizard - File references
Please provide a list of the welcome files used by your WAR file. List any tag libraries referenced by JSPs in this WAR file, and indicate the corresponding tag library. Also, list any errors or exceptions to be trapped and indicate the corresponding resource to be called.
Welcome Files
Add
Delete...
JSP Tag Libraries
Coded Reference
Tag Library
/jsptl-jr
/WEB-INF/jsptl-jr.tld
Add
Delete...
Error Mapping
Error/Exception
Resource to be Called
Add
Delete...
Help...
Cancel
< Back
Next >
Finish

Finally, click Finish. The `jsptl` application now includes the JSPTL Tag Libraries, and you can create new Web Applications that use this library.

Using the JSPTL `forEach` Tag

The JSPTL `forEach` tag performs a similar function to the `forEach` tag you developed today as part of the case study. However, the JSPTL iteration tag supports many different Java idioms for defining lists of values:

- Arrays of Java objects
- Arrays of Java primitives (the individual values are automatically wrapped an the appropriate wrapper object)
- Objects in a `java.util.Collection`, such as list and sets
- Objects in a `java.util.Map` collection
- Objects defined by a `java.util.Iterator`
- Objects defined by a `java.util.Enumerator`
- Objects defined by a `java.sql.ResultSet`
- Comma-separated values in a `String`, such as `"apple,orange,pear"`

The `forEach` tag defines a scripting variable to hold the value for the current loop iteration. You specify the name of this variable with the `var` attribute. The actual list of values is defined by the `items` attribute.

Listing 14.20 shows a simple example that displays a list of fruits defined in a comma-separated string.

LISTING 14.20 Full Text of `foreach.jsp`

```
1: <%@ taglib uri="/jsptl-jr" prefix="jr" %>
2: <HTML><HEAD><TITLE>JSPTL foreach example</TITLE>
3: </HEAD>
4: <BODY>
5: <H1>JSPTL For Each Example</H1>
6: <jr:forEach var="fruit" items='"apple,orange,pear"'>
7:     <%=pageContext.getAttribute("fruit")%><BR>
8: </jr:forEach>
9: </BODY></HTML>
```

The Java string literal is defined as a tag attribute. The attribute is quoted in single quotes, and the string literal is defined in double quotes. The loop iteration variable is stored in the page context and has to be retrieved using the `PageContext.getAttribute()` method, as shown at line 7 of Listing 14.20.

Using the same `forEach` tag, but this time defining the fruits in an array of `String` objects, is shown in the following code fragment:

```
<% String[] fruits = {"apple","orange","pear"}; %>
<jr:forEach var="fruit" items="<%=fruits%>">
    <%=pageContext.getAttribute("fruit")%><BR>
</jr:forEach>
```

You can incorporate the JSPTL `forEach` tag into the case study example. Listing 14.21 shows a simple page to display a list of customers in the `Agency` database.

LISTING 14.21 Full Text of `listcust.jsp`

```
 1: <%@ taglib uri="/jsptl-jr" prefix="jr" %>
 2: <HTML><HEAD><TITLE>JSPTL foreach customer</TITLE>
 3: </HEAD>
 4: <BODY>
 5: <jsp:useBean id="agency" class="web.AgencyBean" scope="request" />
 6: <H1><jsp:getProperty name="agency" property="agencyName"/></H1>
 7: <P>
 8: <jr:forEach var="cust" items="<%=agency.findAllCustomers()%>">
 9:     <%=pageContext.getAttribute("cust")%><BR>
10: </jr:forEach>
11: </BODY></HTML>
```

To deploy the example in Listing 14.21, you will need to add the `Agency` Session bean class files to your application. Figure 14.8 shows the files added to the `jsptl` application, which are `agency.AgencyHome`, `agency.Agency`, `agency.NotFoundException`, `agency.DuplicateException`, and `web.AgencyBean`.

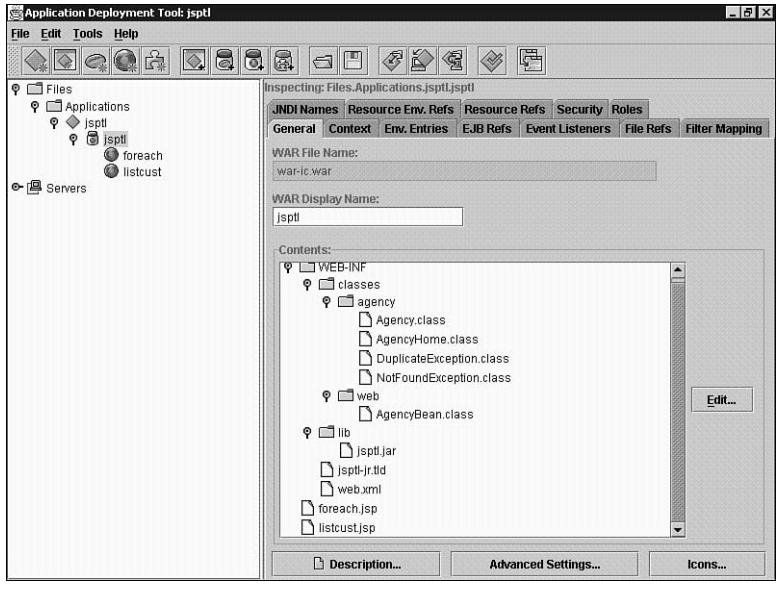


FIGURE 14.8

The `Agency` class files used by `listcust.jsp`.

Other JSPTL Tags

The JSPTL library defines other structural tags. Tags are available to support tokens defined in a `String`, an `if` construct, and a `case` statement. Each of these is shown briefly in this section.

The `forTokens` tag iterates over a delimited list of values specified in a String. The `delims` attribute defines the delimiter, for example

```
<jr:forTokens var="token" items="apple|orange|pear" delims="|">
  ...
</jr:forTokens>
```

Simple optional inclusion of a tag body is supported by the `if` tag that defines a `test` attribute, for example

```
<jr:if test='<%((String)pageContext.getAttribute("cust")).equals("george")%>'>
  ...
</jr:if>
```

A case statement is supported by the `choose` tag and the nested `when` and `otherwise` tags. The `choose` tag structure is as follows:

```
<jr:choose>
  <when test="...">
    ...
  </when>
  <when test="...">
    ...
  </when>
    ...
  <otherwise>
    ...
  </otherwise>
</jr:choose>
```

As you can see, the JSPTL provides the basic structural elements necessary to add programming language, such as constructs to your Web pages.

JSPTL Scripting Language

The support for a scripting language in the JSPTL is very experimental and in the early stages of development. The intention behind the scripting language is to incorporate a powerful technique for adding programming capabilities to tags.

At the time of writing (late 2001), the JSPTL specification provides for pluggable scripting languages. A simple language is defined in the specification, but it should be possible to add other scripting languages, such as JavaScript or Perl.

The following example shows a `forEach` tag that iterates over the values defined by a scripting variable called `customers`.

```
<jx:forEach var="customer" items="$customers">
  <jx:expr value="$customer"/><br>
</jx:forEach>
```

In this example, the Tag Library prefix is `jx` rather than `jr`. The scripting variables are identified by a dollar sign (`$`). The `expr` tag is used to evaluate an expression in the scripting language and writes the results to the Web page (much like the request time `<%= >` tag). Other tags are defined for declaring scripting variables and setting the values of the scripting variables.

Full documentation for the latest developments in the scripting language support are available from the JCP and Apache Jakarta Web pages already mentioned.

Other Jakarta Tag Libraries

The Apache Jakarta open source project is developing a large number of Tag Libraries. Many of these libraries will become the de facto standard for JSP pages.

Already available are libraries for supporting

- Time and Date
- Database access
- I/O support for protocols, such as HTTP, HTTPS, FTP, XML-RPC, and SOAP (SOAP is discussed further on Day 20, “Using RPC-Style Web Services with J2EE,” and Day 21, “Web Service Registries and Message-Style Web Services”)
- JNDI
- Support for HTTP session data
- Regular expressions
- XSL (XML style sheets that are discussed on Day 17, “Transforming XML Documents”)

New Tag Libraries are being developed and added to this collection as a need for them is identified. The Apache Jakarta Web site is the place to look for standard Tag Libraries that may implement the functionality you require.

`http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html`

Summary

Tag Libraries (TagLibs) define one or more custom tags that can be used to extend the capabilities of JSP. Judicious use of TagLibs can remove most, if not all, of the Java scriptlets from your Web pages. Encapsulating the Java code in a custom tag will

- Simplify the writing of Web pages
- Reduce development time by removing awkward Java compiler errors from the JSP-generated code
- Integrate business logic constructs more closely into the JSP syntax

Custom tags can provide programming constructs, such as iteration and selection (`if` statements) for your Web pages. Custom tags can also

- Use attributes to pass information from the Web page to the Java code
- Create scripting variables that can be used on the Web page
- Share information between tags in a hierarchical structure

Standard Libraries, such as JSPTL and the Apache Jakarta libraries, provide many useful tags that can quickly be incorporated into your Web applications. Using existing libraries reduces the amount of code you have to develop, and that will reduce the overall development time for your application.

Q&A

Q What are three features of custom tags that I can use to extend my Web pages?

A Custom tags can access objects (such as HTTP request parameters and the page context) of the JSP page. They can cooperate and share information with each other and can be used in a hierarchical manner to implement programmable concepts, such as iteration.

Q What is the name of the Java package that supports the development of custom tags, and what are the two classes that are normally extended by custom tags?

A The custom tag package is `javax.servlet.jsp.tagext`, and the two super classes used for most tags are `TagSupport` and `BodyTagSupport`.

Q What are the five methods I can override when extending `BodyTagSupport`, including parameters and return type? In what order are these methods call when processing the custom tag?

A The `BodyTagSupport` methods in execution order are

1. `int doStartTag()`
2. `void doInitBody()`
3. `int doAfterBody()`
4. `int doEndTag()`
5. `void release()`

Q What does TLD stand for and what is it used for?

A TLD is the Tag Library Descriptor, and it is an XML document that defines the Tag Library. The TLD is used to define the library name and version of the JSP specification to which your tags conform. It also lists every tag in your library.

Q What are the five XML components that are used in the custom tag `<tag>` entry in the TLD?

A The most common tags used in the `<tag>` entry in a TLD are

1. `<name>` The tag name
2. `<tag-class>` The implementing Java class
3. `<body-content>` How the tag body is defined
4. `<attribute>` Defines an attribute for the tag
5. `<variable>` Defines a scripting variable created by the tag

Exercise

Refactor the `register.jsp` page you developed on Day 13 to use custom tags. Write a `getApplicant` tag that provides support for accessing the `Applicant` Session bean and remove the need for the `web.ApplicantBean` class. Model your code on the changes to the `advertise.jsp` page and the new `getCust` tag you were shown in today's work. As a starting point for the code for your exercise, use the agency code from today's `examples` sub-directory of the case study on the accompanying CD-ROM. The existing `agency.jsp` page includes the code for creating a new applicant and for selecting an existing applicant for update or deletion.

You will need to create the following JSP files to support the register applicant functionality:

- `register.jsp`
- `createApplicant.jsp`
- `deleteApplicant.jsp`
- `updateApplicant.jsp`

You can use the versions of these files from Day 13 and modify them to use the tag libraries you developed today for handling locations and skills. In addition, you will need to develop a new custom tag to create an instance of a `Register` Session bean for use on the Web page. Call the new Tag Library class `GetAppTag.java`.

Model your implementation and use of this tag following the example `GetCustTag` shown in this chapter.

Don't forget to add an entry for the `GetApp` tag to the `agency.tld` file.

A solution to this exercise is incorporated in the Agency case study stored in the `agency` sub-directory of Day 14 on the accompanying CD-ROM.

Week 3

Integrating J2EE into the Enterprise

15 Security

16 Integrating XML with J2EE

17 Transforming XML Documents

18 Patterns

19 Integrating with External Resources

20 Using RPC-style Web Services with J2EE

21 Web Service Registries and Message-style Web Services

WEEK 3

DAY 15

Security

So far, you have developed your J2EE application without considering security. Now you will look at how to add security constraints to your system to prevent loss of privacy or to keep unauthorized clients from accessing data and causing accidental or malicious damage.

In today's lesson, you will look at

- How the J2EE specification supports the common requirements for a secure system
- The common terminology used when discussing system security
- Symmetric and asymmetric encryption
- Securing a J2EE application using principals and roles
- Using declarative security for EJBs and Web pages
- Using programmatic security in EJBs and Web pages
- Supplying security credentials to an LDAP naming service provider for JNDI

Security Overview

Security is an essential aspect of most, if not all, enterprise applications. However, defining an application as secure is not as easy as it sounds, because the definition of secure can be interpreted in different ways.

To some users, a Web site is secure if they have to provide a username and password to obtain access to the Web pages. As you will see, just because a site requires a user to login does not make it secure.

Security Terminology

Security has many aspects that can be categorized into the following areas:

- Authentication
- Authorization
- Confidentiality
- Integrity
- Non-repudiation
- Auditing

Each of these categories is discussed in this section.

Authentication

Authentication means identifying a client as a valid user of the system. Identifying a client has two components:

- Initially confirming the client's identity
- Authenticating the client each time it accesses the application

Initial Identification

At its simplest level, initial identification requires a user to simply register with an application without any additional identification. More often, a third party, such as the Human Resources department or manager in a company, identifies a user. At its most complex level, usually associated with military systems, identification requires background checks to confirm a user's identity. Identified users are registered with the system and granted access to some or all of the facilities provided by the system (see the "Authorization" later in this chapter).

Client Authentication

Registered users of an application must identify themselves each time they use the application. The most common form of authentication is to give each user a unique name (typically an account or login name) and a password associated with that account. Users simply have to provide their account names and passwords to gain access to the application.

The information identifying a client is usually called the *user credentials*. The most commonly encountered forms of user credentials are as follows:

- Account name and password
- Swipe cards
- Smart cards
- Physical identification systems, such as fingerprints and retinal images
- Digital certificates

Authentication is like the entrance gate to a modern theme park. As long as you have a ticket, you are allowed into the park—you have been authenticated. But authentication does not necessarily allow you to use all of the rides and facilities in the park. The means by which you are allowed access to different parts of the theme park is called *authorization*.

Authorization

Authorization involves controlling access to capabilities of an application according to the authenticated user's identity. Authorization differentiates between different categories or types of users and grants, or denies them access to different parts of the system.

Using the theme park analogy again, you may only be authorized to use certain rides. Rides may have height, weight, or age restrictions that authorize access to some users and deny access to others.

Confidentiality

Another aspect of security relates not to controlling access to functionality but to ensuring that data is only seen by authorized users. In other words, the data remains confidential. Maintaining confidentiality is not just a question of authorizing access to the data but also of ensuring unauthorized access either cannot occur, or if it does, that the data remains "secure." In practical terms, confidentiality is often achieved by encrypting the data so that only authorized users can decrypt and access the data.

Integrity

Ensuring data integrity means preventing deliberate or accidental attempts to modify the data in an unauthorized manner. Applying authorization correctly solves most of the data integrity problems concerned with accessing data on a server.

Data transferred across the network must not be changed or corrupted as it is transferred. The user must be sure that the data they receive is the data that was transmitted. Techniques, such as encryption, checksums, and message digests (see the "Messages Digests and Checksums" section later in today's lesson), help ensure data integrity across networks.

Integrity also means that any changes made to a system, are not lost, such as might occur when a server crashes. Good auditing practices (see the "Auditing" section later in this chapter) help prevent the loss of changes to persistent data.

Non-Repudiation

Non-repudiation means being able to prove a user did something, even if the user subsequently denies it. A simple example is a user with online banking facilities. A fraudulent user could transfer money to another bank account and then try to claim this was a spurious transaction and a fault of the banking system. With good accounting processes, the bank can prove this was not the case.

Auditing

Auditing is familiar to database users and has the same meaning in security—providing a record of activity. Good auditing is an adjunct to supporting non-repudiation and integrity. Remember, audit records must themselves be kept secure.

Common Security Technology

Modern software architectures make use of several technologies for supporting system security. This section is a quick summary of the key technologies that are used and how then can help support the different aspects of secure systems.

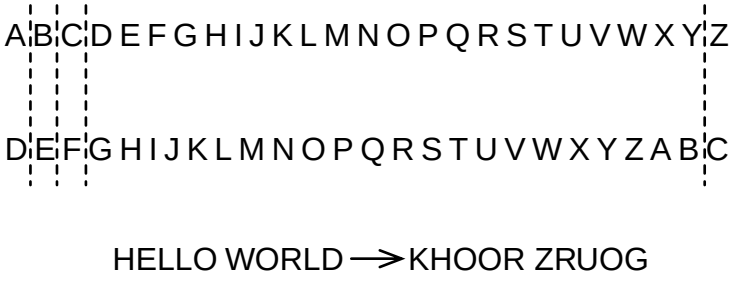
Symmetric Encryption

Data encryption means converting the data so that it can only be decrypted and read by authorized users. Data encryption requires an algorithm that is applied to the data to encrypt it.

Symmetric encryption is so called because it uses the same key to both encrypt and decrypt the data.

One of the simplest cryptographic techniques is the Caesar cipher (named because Julius Caesar was reported to have used it). The Caesar cipher simply replaces each letter of the alphabet by the letter three positions further on, so that A is replaced by D, B by E, and so on, with the last three letters replaced by the start of the alphabet. Figure 15.1 shows the Caesar cipher.

FIGURE 15.1
The Caesar cipher.



The Caesar cipher is a specific example of a simple shift substitution cipher when one letter replaces another. A different cipher is obtained by shifting the alphabet by more or less than three letters, as shown in Figure 15.2.

FIGURE 15.2
A Shift cipher.

KEY
0 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
2 C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
3 D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
25 Z A B C D E F G H I J K L M N O P Q R S T U V W X Y

The number of letters the enciphering alphabet is shifted by is called the *cipher key*. Given an encrypted message, anyone with the key can decipher the message. Because the same key is used to encrypt and decrypt the message, this is know as a *symmetric encryption algorithm*.

In programming terms, each letter is represented by a number, and the substitution cipher simply adds the key number to the value of each letter to get the encrypted form. The resultant number must be adjusted to map the last few letters (X, Y, and Z) onto the first few (A, B, and C) letters. This is a very simple algorithm.

In real applications, symmetric algorithms use sophisticated algorithms with number keys of 56 or 128 bits (approximately 45 decimal digits). The algorithms used are usually well known but, due to the size of the keys used, they cannot be easily reversed. In other words, without the key, the original plain text message can only be recovered by applying each possible key in turn. As long as the key is a large one and the encryption algorithm is sufficiently robust, the time taken to crack the cipher with a brute-force method attack, such as applying every possible key, can be hundreds of years.

One of the most widely used symmetric encryption algorithms is called DES (Data Encryption Standard).

Symmetric encryption is used to ensure data confidentiality. Symmetric encryption ensures that only the intended recipients who know the decryption key can recover the original data.

Asymmetric Encryption

Asymmetric encryption uses different algorithms than symmetric encryption and requires the use of two keys. One key is used to encrypt the data, and the other is used to decrypt the data. The two keys can be very large numbers, with modern systems using numbers of 1024 bits (approximately 140 decimal digits). Asymmetric encryption is called *public key encryption* due to the way the two keys are used.

Of the two keys used in asymmetric encryption, one is made public, while the other is kept private to the owner. The keys are known, respectively, as the *public key* and the *private key*.

If data is encrypted with the public key, only the owner of the private key can decrypt it. This approach is used to ensure data confidentiality but is restricted to supporting only one recipient per message.

In contrast, using symmetric key encryption allows one message to be distributed to several recipients, as long as each recipient knows the key used to encrypt the message. If the private key was known by more than one person, it would undermine the other benefits of using asymmetric encryption (such as non-repudiation) and avoid the need to distribute the cipher key. Distributing the keys used in symmetric encryption is a major problem, because the keys have to be distributed in a secure manner. An attacker obtaining the keys can decrypt the message to recover the original data.

Another use of asymmetric encryption is to support non-repudiation. If a message is encrypted with the private key, it can only have originated from the key owner. Anyone can decrypt the data using the public key with the knowledge that it can only have originated from the owner of the private key. This use of asymmetric encryption is the basis of *digital signatures*.

Asymmetric encryption is slow compared to symmetric encryption. To improve performance, it may be desirable to use symmetric encryption. The problem here is how to distribute the encryption key to the recipient securely.

A common approach is to use symmetric encryption for the data and to pass the encryption key with the data. To make this approach secure, the recipient's public key is used to encrypt the symmetric key passed with the encrypted data. The recipient can use his or her private key to recover the key and then decrypt the actual data.

This technique enables large volumes of data to be encrypted quickly and, at the same time, distributing the encryption key in a secure manner.

SSL and HTTPS

The Secure Sockets Layer (SSL) is an implementation of public key encryption in TCP/IP networking. TCP/IP communication uses a technology called sockets (sometimes called service or port numbers). All standard TCP/IP services advertise themselves on a fixed socket or port—FTP on 21, TCP/IP on 23, HTTP on 80, and so on. You have seen socket numbers when using the J2EE RI Web server that runs on port 8000.

```
http://localhost:8000
```

Ordinary socket communication uses plain (unencrypted) data. Any user that can monitor network traffic can read any usernames, passwords, credit card details, bank account information, or anything else passed over the network. This is obviously an unacceptable situation from a security point of view.

One solution to securing confidential data over a network is to encrypt the data within the application. This is an inconsistent solution because some applications will be secure while others are not.

Another solution is to always encrypt all network traffic. Because encryption adds an overhead to the network communication, this will affect overall performance and is unnecessary when data does not need to be encrypted.

The workable solution is to seamlessly provide network encryption only for applications that require secure data transmission. Using this approach, any application can encrypt confidential data simply by using the encrypted network communications instead of the usual plain text data transfer. Each application decides if encryption is needed but does not have to implement the encryption algorithms.

SSL is a network encryption layer than can be used by any TCP/IP application. The application has to connect by using a secure socket rather than a plain socket, but otherwise, the application remains unchanged.

Hypertext Transfer Protocol Secure (HTTPS) is the name given to the HTTP protocol when it uses a secure socket. The default port used by an HTTPS is 443. When a URL specifies the HTTPS service, the Web browser connects to an HTTP server but uses SSL to encrypt the data. All the popular Web browsers indicate on the status line when SSL communication is taking place. Typically, an open and closed padlock is used to shown whether data is encrypted.

Online credit card verification services and banking systems use SSL communication.

Checksums and Digests

Data integrity is usually achieved by providing checksums or digests of the data. The data in a message is subjected to a numerical algorithm that calculates one or more validation numbers that are transmitted with the data. The recipient receives the data and applies the same algorithms to the data. As long as the recipient's calculations yield the same numbers as those transmitted with the data, the recipient is reasonably confident that the data is unchanged.

Checksums use simple algorithms and are primarily intended to detect accidental corruption of data. Message digests use sophisticated algorithms that are designed to prevent deliberate changes to data. The algorithms used in a message digest generate many digits and are chosen so that it is virtually impossible to change the original data without changing at least one of the digest numbers.

There are several digest algorithms in use, with Message Digest version 5 (MD5) currently one of the most popular. The MD5 specification can be found at `http://www.ietf.org/rfc/rfc1321.txt`.

Many applications that can be downloaded from the Web also have an associated signature file. A signature file is used to validate the contents of the associated file (the one it signs). Signature files usually contain one or more digests (typically MD5) of the file they are signing. After downloading the file, a conscientious user can also download the signature file and check the integrity of the download file by calculating the digest of the file and comparing it to the value in the signature file. Programs to calculate digests are widely available on the Internet.

Digital Certificates

Digital certificates are specified by the X509 international standard and define a format for representing public keys and other information about an entity (it could be a user, a program, a company, or anything that has a public key).

The official specification for the X.500 Directory Service is available from the International Telecommunications Union (ITU) web site at `http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=TREC-X.509`.

Digital certificates are often sent with a request for data so that the server can encrypt the data with the recipient's public key.

Digital certificates must be signed by a *Certification Authority* (CA) to prove their validity. A signed digital certificate contains a message digest of the certificate encrypted using the CA's private key. Any recipient of the certificate can decrypt the digest using the CA's public key and verify that the rest of the certificate has not been corrupted or modified.

Digital certificates can be used to ensure authentication, confidentiality, and non-repudiation.

Valid Digital Certificates have been erroneously issued to individuals spoofing the credentials of trusted companies. A Digital Certificate is only as trustworthy as its Certification Authority.

Security in J2EE

The J2EE specification takes a pragmatic approach to security by focusing primarily on authorization within the J2EE environment and integration with security features that already exist in the enterprise.

You have already seen the J2EE design philosophy of separating roles with the development lifecycle identifying code developers, application assemblers, deployers, and administrators. The J2EE security supports this role-based model by using two forms of security:

- *Declarative security*—Declarative security is defined within the application's deployment descriptor (DD) and authorizes access to J2EE components, such as Web pages, servlets, EJBs, and so on. End user tools, such as the J2EE RI `deploytool`, support declarative security.
- *Programmatic security*—Programmatic security is used when declarative security is not sufficient to meet the needs of an application. Security-aware components implement the security requirements by using programming constructs.

The J2EE security specification also requires transparent propagation of security credentials between components. In layman's terms, this means that once clients have logged in to a Web page, they do not need to authenticate themselves again for any EJBs accessed from the Web page. Also, the authenticated identity of the user remains the same for all components (Web pages, servlets, client applications, and EJBs).

J2EE Security Terminology

The J2EE security domain is based around the concept of principals, roles, and role references.

Principals

Principals represent authenticated entities, such as users. The authentication mechanism is not defined within the J2EE specification, allowing existing authentication schemes to be integrated with a J2EE application.

The downside of not defining how users are authenticated means that some parts of the J2EE security features vary between one manufacturer's implementation and another. As the J2EE specification has evolved, additional security requirements have been incorporated to remove the variation between implementations. Future versions of the J2EE specification will add additional security-related constraints as the underlying technology matures and standardizes.

In a simple implementation, a J2EE principal is a user and the principal's name is the username. However, there is no requirement for a particular implementation to map the real usernames onto unique principal names. In fact, a principal can represent a group of users rather than an individual user. Principal names are only used in programmatic security and are inherently non-portable.

Wherever possible, J2EE security should be based on roles rather than principals because roles are more portable than principal names. Using principals to implement security requires coordination between the developer and the deployer and can restrict the reusability a particular J2EE component.

Roles

Roles are identified by the developer and represent how components in an application will be used. Typically, a developer will identify roles, such as user, administrator, manager, and so on, and suggest how the functionality in the application will be used by each identified role.

A deployer will map principals (real users and groups of users) onto the roles defined in the application. The deployer has total control on how the actual security authentication is mapped onto the J2EE application.

An assembler will combine the roles from many different components to create roles that represent common security requirements across different components.

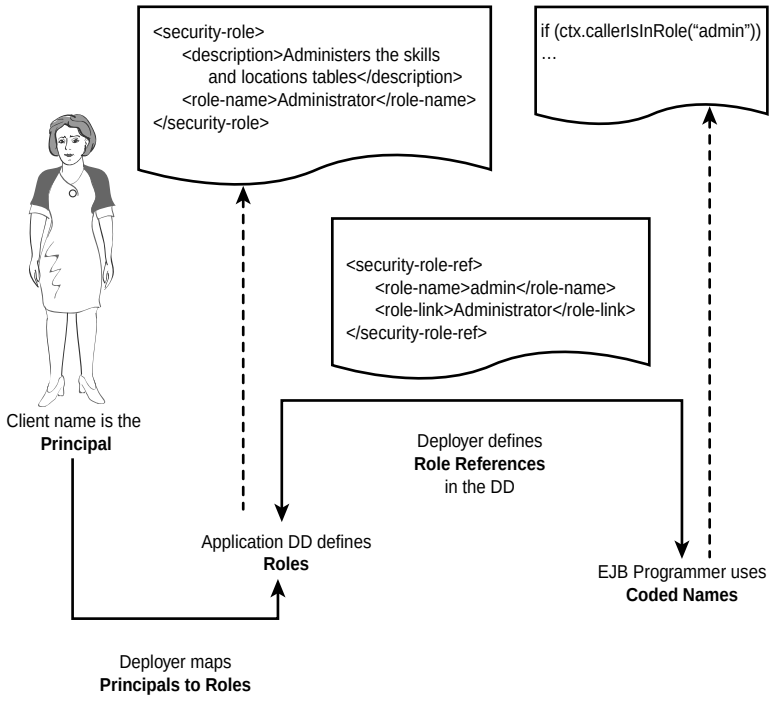
Role References

Role references are used to map the names used in programmatic security to the roles defined in the deployment descriptor (DD). The developer defines the coded role names used in the code, and the assembler or deployer maps the coded references onto the roles defined in the application.

The relationships between principals, role references, and roles are shown in Figure 15.3.

FIGURE 15.3

Mapping J2EE principals and roles.



Working with J2EE RI Security

So far, you have only been aware of J2EE RI security when running client applications. Even though your applications so far have not been security aware, the J2EE RI environment requires your code to be run inside an environment that has a security context.

Web applications run within the J2EE RI Web server that does have a security context. However, applications that run from the command line, such as those you developed on Day 4, "Introduction to EJBs," and Day 5, "Session EJBs," do not have a security context. You must add your client classes to the Enterprise Application Resource (EAR) file as a client application to obtain a security context. The `runclient` program is used to run your client applications with a security context.

Before you look at making your J2EE application security aware, you must spend a short time looking at the J2EE RI support for a simple authentication system.

The J2EE RI provides an authentication domain that can be used during application development. The RI security domain supports:

- *Realms*—A realm defines users that are authenticated using the same mechanism. The J2EE RI defines two realms:
  - `default` Consisting of users identified by passwords
  - `certificate` Consisting of users identified by X509 digital certificates, (certificates are only used to authenticate Web browser clients)
- *Users*—Defines a username within the J2EE security domain. In the default realm, the username is the principal name. In the certificate realm, the common name on the certificate is the username.
- *Groups*—Users in the default realm can be assigned to groups. Groups can be mapped onto role references to simplify security administration.

Adding Users and Groups

Users and groups can be added to the J2EE RI server via two tools:

- `realmtool` A command-line–based interface
- `deploytool` A GUI interface

Both tools provide limited support for adding users, groups, and certificates to the J2EE authentication domain. After changes have been made to the J2EE security realms, the server must be restarted before those changes take effect.

The command line `realmtool` supports the following options:

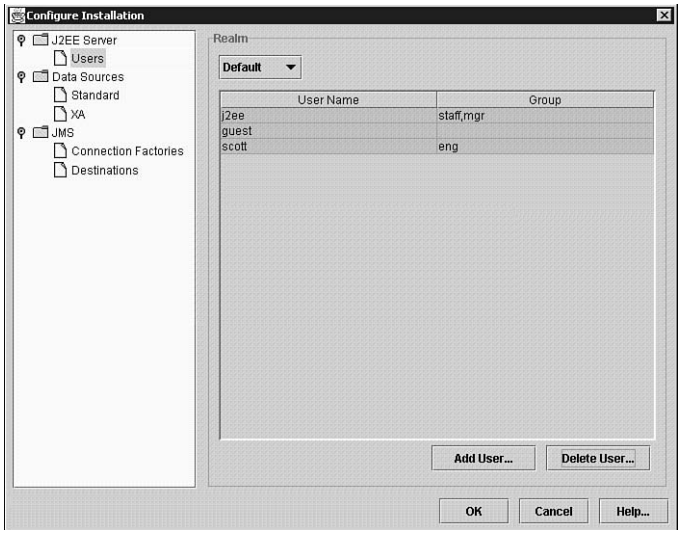
```
RealmTool
Options
 -show
 -list          realm-name
 -add           username password group[,group]
 -addGroup      group
 -import        certificate-file -alias alias
 -remove        realm-name username
 -removeGroup   group
```

The tool lacks the ability to list groups in the default realm, but it has the advantage of working without having to have the J2EE server up and running.

The GUI interface is more functional and easier to use and is accessed via the "`Tools -> Server_Configuration`" menu in `deploytool`. Figure 15.4 shows the main user configuration screen.

Although the GUI interface is easier to use, it requires the J2EE server to be running. The server must be stopped and restarted for the changes to be recognized by your applications.

FIGURE 15.4
Adding users with `deploytool`.



Neither utility supports the ability to change a user's password. If a password is forgotten, the user must be deleted and re-created to define a new password.

The default users provided with the J2EE RI server are shown in Table 15.1.

TABLE 15.1 Pre-Defined J2EE RI Users

User	*Password*	*Groups*
`j2ee`	`j2ee`	`staff, mgr`
`guest`	`guest123`	
`scott`	`tiger`	`eng`

Both utilities are intuitive to use. For today's work, you will need to add some sample users as defined in Table 15.2.

TABLE 15.2 Agency Case Study Users

User	*Password*	*Group*
`romeo`	`romeo`	`Applicant`
`juliet`	`juliet`	`Applicant`
`winston`	`winston`	`Customer`
`george`	`george`	`Customer`

You will need to add the groups for `applicant` and `customer` as well as the users shown in Table 15.2. After making your changes, don't forget to stop and restart the J2EE server.

You must make these changes to your J2EE server if you want to use the example code provided on the accompanying CD-ROM.

Security and EJBs

EJB security is determined either by the declarative entries added to the DD, the programmatic constraints coded into the EJBs, or a combination of both.

Ideally, EJB security should only use the declarative approach, but where declarative security cannot represent the application's requirements, security must be encoded in the EJB class. The programmatic security is less portable and may restrict the way an application assembler can combine beans from different sources.

Defining EJB Security

Defining security for an EJB involves

- Defining one or more roles to control access to different areas of your application
- Restricting access to EJBs and EJB methods according to the clients roles
- Mapping roles onto principals in the authentication domain
- Optionally adding programmatic authorization to Session and Entity beans

If you are using the J2EE RI, the security can be defined using `deploytool`. Other J2EE environments may provide GUI tools similar to `deploytool` or, if you are unlucky, you may have to manually edit the DD to include the security requirements.

In the rest of this section, you will use `deploytool` to add security to the Agency case study. You will see how extra information is added to the DD to define the security requirements.

Defining Roles

There are three distinct roles with the simple `Agency` application:

- Administrators that can modify the skills and location tables
- Customers who advertise jobs
- Applicants who advertise their locations and skills

You may even decide that there are only two roles—administrators and clients (for want of a better term). Clients can register their own skills or advertise jobs for other clients. While this is a perfectly acceptable model, it looses the differentiation between applicants and customers.

Currently, there are no constraints on who can be a client and who can be a customer. However, in a real world job agency, it may become necessary to restrict who can be applicants and customers. Perhaps customers will be charged for applicants who match their jobs, so they need to be validated before they can use your system.

Having decided on your roles, you must add them to the DD. Roles are added to JAR files in your application. If several EJBs are defined in the same JAR file, they can share the same roles. EJBs in separate JAR files must define their own roles. If the same role name is used in different JAR files, it still represents a different role.

Grouping related EJBs into a single JAR file is a good design philosophy because it allows related beans to share the same security roles.

Now you will add security constraints to the Agency case study. If you want to look at the finished results of following the steps described in the rest of this section, you can look at the `agency.ear` file in the `exercises` directory for Day 15 ("Security") on the accompanying CD-ROM.

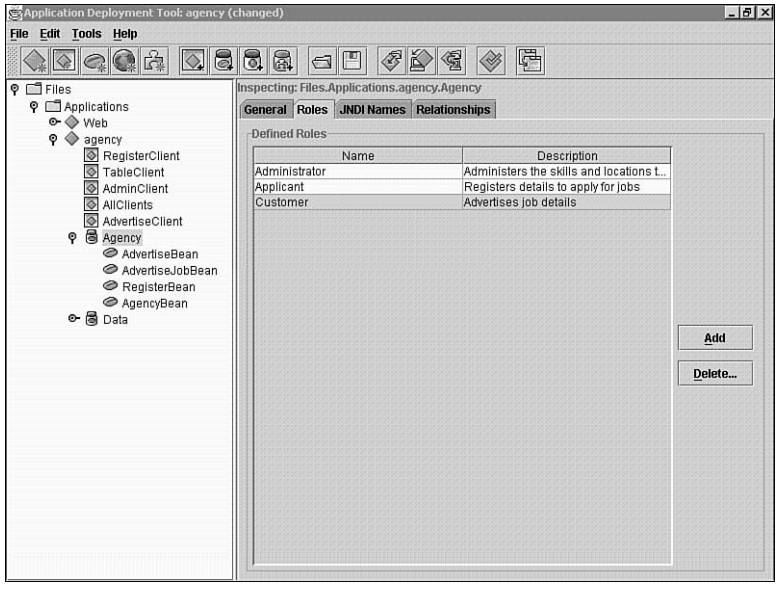
If you want to add security to the Agency case study as you have developed it so far, start up `deploytool` and open the `agency.ear` file in the Day 10 ("Message-Driven Beans") `examples` directory. The `Agency` application has three JAR files—one for the Session beans, one for the Entity beans, and one for the Web interface. Select the `agency` Session bean and display the "`Roles`" page. Use the `Add` button to add the three roles defined in Table 15.3. Figure 15.5 shows the screen after adding these roles.

TABLE 15.3 Agency Case Study Roles

Role	*Description*
`Administrator`	Administers the skills and locations tables
`Applicant`	Registers details to apply for jobs
`Customer`	Advertises job details

Roles are defined in the `<assembly-descriptor>` component inside the `<ejb-jar>` tag in the DD. Listing 15.1 shows the entry created for the three roles you have just defined.

FIGURE 15.5
Adding security roles.



LISTING 15.1 Security Role Entries in the DD

```
1: <assembly-descriptor>
   2:    <security-role>
   3:      <description>Administers the skills and locations tables</descrip-
tion>
   4:      <role-name>Administrator</role-name>
   5:    </security-role>
   6:    <security-role>
   7:      <description>Registers details to apply for jobs</description>
   8:      <role-name>Applicant</role-name>
   9:    </security-role>
  10:    <security-role>
  11:      <description>Advertises job details</description>
  12:      <role-name>Customer</role-name>
  13:    </security-role>
  14: </assembly-descriptor>
```

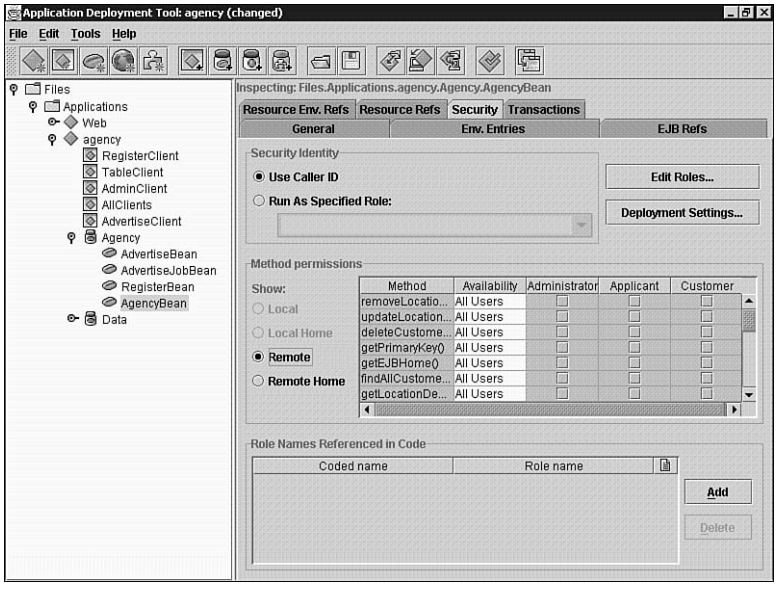
A `<security-role>` tag with `<role-name>` and `<description>` elements defines each role.

Defining the Security Identity

After the roles for a JAR file have been defined, you can restrict access to the methods of an EJB. In `deploytool`, select the `AgencyBean` EJB and then the Security tab. You will see the screen shown in Figure 15.6.

FIGURE 15.6

Security for EJB methods.



The `Security Identity` section at the top of the screen shows that authorization is controlled by the caller's identity. There are two options for security identity:

- Use the caller's ID.
- Use a defined role.

In Figure 15.6, Use Caller ID is selected.

The `<security-identity>` tag in the DD defines how access to an EJB is authorized. The tag is part of the `<session>` bean definition, and the `<use-caller-identity>` option is shown in Listing 15.2.

LISTING 15.2 Security Identity Entries in the DD

```
 1: <enterprise-beans>
 2:   …
 3:   <session>
 4:     <display-name>AgencyBean</display-name>
 5:     <ejb-name>AgencyBean</ejb-name>
 6:     <home>agency.AgencyHome</home>
 7:     <remote>agency.Agency</remote>
 8:     <ejb-class>agency.AgencyBean</ejb-class>
 9:     <session-type>Stateless</session-type>
10:     …
11:     <security-identity>
12:       <description></description>
13:       <use-caller-identity/>
```

LISTING 15.2 Continued

```
14:      </security-identity>
15:      …
16:    </session>
17:    …
18: </enterprise-beans>
```

The use of roles for the security identity is discussed in the "Using Roles as the Security Identity" section later in this chapter.

The lower part of the `deploytool` Security page is the `Role References` section, and this is discussed in the section "Programmatic EJB Security".

Defining Method Permissions

The `Method Permissions` section in the middle of the `deploytool` window shown in Figure 15.6 lists the methods for the interfaces defined for your bean. The radio button selects which interface is displayed (the default for the `agency` bean is the remote interface). Each bean method defines a row in a table, and the columns are the roles defined for the JAR file. By using the cells in this table, you can select which methods can be called by each role.

From Figure 15.6, you can see that the default access (the Availability column) to all methods is All Users which is why, so far, you have been able to access all your application functionality regardless of any username you supplied when logging in to run the application.

In the underlying DD, the method permissions are added to the `<assembly-descriptor>` tag. The `<method-permissions>` tag associates one or more permissions with one or more methods. A permission is the name of a role specified with the `<role-name>` tag or the empty tag `<unchecked/>` to show that access is unchecked (it is callable by all clients). A method is defined by the `<method>` tag that has three variants:

1. Authorize all methods in an EJB using a tag of the following form:

   ```
   <method>
     <ejb-name>EJBname</ejb-name>
     <method-name>*</method-name>
   </method>
   ```

   where the * means all methods

2. Authorize a named method in an EJB using a tag of the following form:

   ```
   <method>
     <ejb-name>EJBname</ejb-name>
     <method-name>MethodName</method-name>
   </method>
   ```

3. Authorize a specific overloaded method in an EJB using a tag of the following form:

```
<method>
  <ejb-name>EJBname</ejb-name>
  <method-name>MethodName</method-name>
  <method-params>
    <method-param>ParameterClass1</method-param>
    ...
    <method-param>ParmeterClassN</method-param>
  </method-params>
</method>
```

The third form is considered unnecessary because there should be no need to differentiate between overloaded functions for security purposes. Overloaded forms of the same method should perform the same function and therefore require the same security permissions. If the security requirements are different, good design would imply using different method names.

The `<method>` tag also allows a `<method-intf>` (method interface) tag for defining the interface name if it is duplicated in the home and remote interfaces.

As an example, you could set all the methods in the `agency` bean to be accessible only by the `Administrator` role using the example in Listing 15.3.

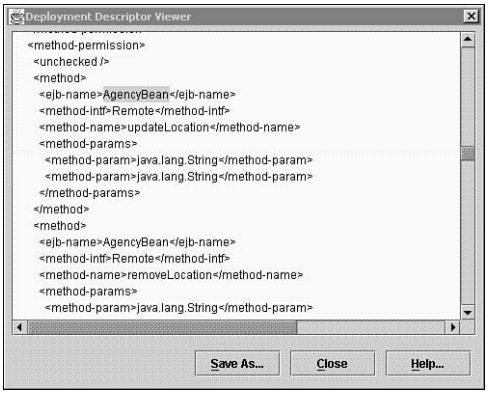
LISTING 15.3 Method Permission Entries in the DD

```
1: <method-permission>
2:   <role-name>Administrator</role-name>
3:   <method>
4:     <ejb-name>agency</ejb-name>
5:     <method-name>*</method-name>
6:   </method>
7: <method-permission>
```

If a method is not listed in a `<method-permission>` tag, that method cannot be called by the client code.

Hopefully, you won't have to write the DD entry yourself but will be able to use a utility, such as `deploytool`. Returning to the `deploytool` screen shown in Figure 15.6, you can see that the default access for all methods is All Users. The All Users permission maps onto the `<unchecked/>` tag in the DD. If you examine the DD using the `"Tools->Descriptor Viewer"` menu, you will see the method permission entries for the agency EJB, as shown in Figure 15.7.

FIGURE 15.7
Security DD for EJB methods.



Applying security is now just a matter of deciding which roles can call which methods for every EJB in your application. In the `agency` bean, you can set the access permissions as shown in Table 15.4.

TABLE 15.4 Agency EJB Authorization

Method	*Roles*
`removeLocation`	`Administrator`
`updateLocation`	`Administrator`
`deleteCustomer`	`Administrator`, `Customer`
`getPrimaryKey`	none
`getEJBHome`	none
`findAllCustomers`	All
`getLocationDescription`	All
`findAllApllicants`	All
`removeSkill`	`Administrator`
`getSkills`	All
`getAgencyName`	All
`getHandle`	none
`select`	`Administrator`
`addLocation`	`Administrator`
`remove`	All
`updateSkill`	`Administrator`
`isIdentical`	none

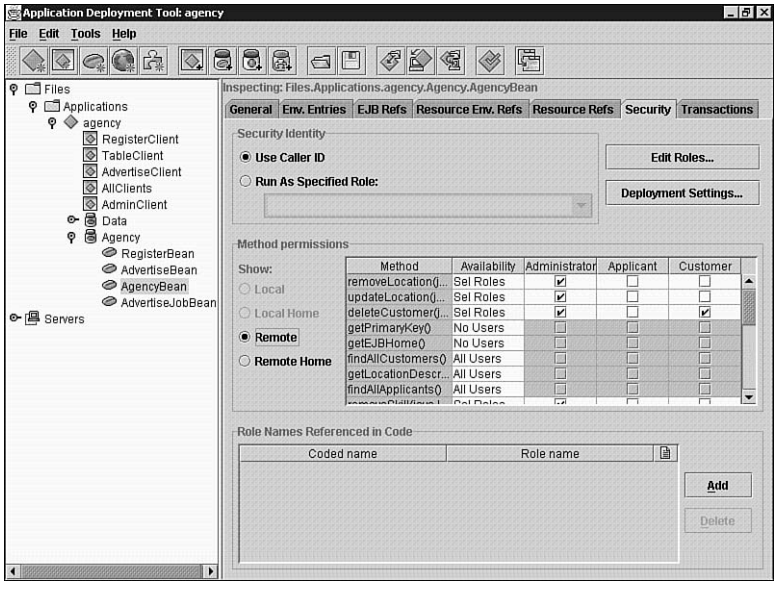
TABLE 15.4 Continued

Method	*Roles*
`addSkill`	`Administrator`
`getSkillDescription`	All
`createCustomer`	`Administrator`, `Customer`
`getLocations`	All
`deleteApplicant`	`Administrator`, `Applicant`
`createApplicant`	`Administrator, Applicant`

Setting the method permissions by using `deploytool` is laborious but, unfortunately, you have no choice in using the tool because you cannot edit the descriptor manually unless you use batch files or shell scripts to rebuild the EAR file as discussed on Day 5. Sometimes manually editing the DD is preferable to using a GUI tool.

When you have added the permissions, your application will look similar to that shown in Figure 15.8.

FIGURE 15.8
Security DD for agency EJB.



Authorization is only required for the methods in the remote interface. Some of the methods (such as `getSkills()`) can be accessed by all users, so the home interface methods (such as `create()`) must also be available to all users. Consequently, the default authorizations are appropriate for the home interface.

This is a slightly contrived example used to illustrate the purpose of roles within an application. In a real world job agency, you would probably only want administrators to create customers. However, you would probably be comfortable with allowing any user to register themselves as an applicant. There would be no need to differentiate between applicant clients and customer clients. In the "Programmatic EJB Security" section later in this chapter, you will see how the Agency case study can use programmatic security to provide a more realistic authorization mechanism.

The remaining three EJBs in the `Agency` application must also have method permissions defined. These are much easier to specify.

The `Register` EJB must have all the home and remote interface methods accessible to the `Administration` and `Applicant` roles only.

The `Advertise` EJB and `AdvertiseJob` EJB must have all the home and remote interface methods accessible to the `Administration` and `Customer` roles only.

After all these method permissions have been defined, the application is now ready for the deployer to map the principals on to the roles you have just defined. If you haven't already done so, it is worth saving your `agency.ear` file at this point just in case something goes wrong later.

Just before you learn how to do this with the J2EE RI `deploytool`, you should consider the security of the Entity beans in the `data` JAR file. How can the Entity beans be accessed, and what authorization is required?

The Entity beans are only ever accessed via the Session beans. Consequently, by applying authorization to the Session beans, you have also protected the Entity beans. Should you redesign the application so that the Entity beans are exposed to client code (maybe by accessing the beans directly from a Web page), you will need to change the method permissions for all of the Entity beans.

By designing access to Entity beans via the Session beans, you have simplified the design of the security features. This is one of the advantages of not exposing Entity beans to client code.

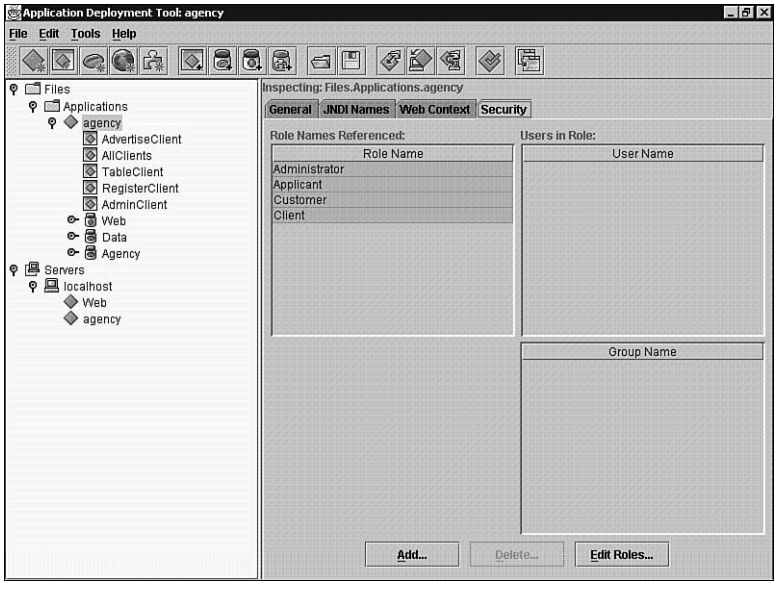
Mapping Principals to Roles

A developer or assembler in the EJB development lifecycle undertakes the process of defining the roles and method permissions. The process of mapping principals to roles is very much a deployer function.

The exact mechanism of mapping principals onto roles is not defined in the EJB specification (at least not yet). There are no tags defined in the DD; instead, each J2EE server defines its own mechanism for mapping principals to roles. The J2EE RI defines a proprietary XML file (`sun-j2ee-ri.xml`) that is stored in the Enterprise Application Resource (EAR) file. This file is maintained by using `deploytool` and deployed with the EJB components.

The `deploytool` maps principals to roles at the application level. To map the roles for the Agency case study, you must run `deploytool` and select the `Agency` application containing the JAR file in which you defined the roles. Select the Security tab and you will see a window similar to the one shown in Figure 15.9.

FIGURE 15.9
J2EE RI principal mapping screen.



You can see that `deploytool` has retrieved the role names from all the JAR files in the application. By clicking the Add button, you can map each role onto one or more users or groups in the authentication domain.

The Agency case study requires the role mappings shown in Table 15.5.

TABLE 15.5 Case study Role Mappings

Role	*User*	*Group*
`Administrator`	`j2ee`	
`Applicant`		`applicant`
`Customer`		`customer`

After defining the role mappings for the Agency case study, you can deploy and test your changes by using the GUI client code in the Day 15 "`examples`" directory.

If you run the example GUI clients from Day 15 on the accompanying CD-ROM, you will see an error message and a stack trace prior to being prompted for a username and password. The last few lines of the error are

```
java.lang.NullPointerException
No local string for enterprise.deployment.unabletoloadtld
Unable to load TLD WEB-INF/agency.tld
 java.lang.Exception: parsing error: null
```

This error can be ignored because it is related to the TLD descriptor used by the custom tag libraries of the Web interface (see Day 14 "JSP Tag Libraries"). This error appears to be a problem with Sun Microsystem's J2EE RI and does not affect the functionality of the Web interface to the Agency case study (you may have seen this exception when working on the examples and exercise for Day 14).

Because the `j2ee` user has unrestricted access, you can test all the functionality if you log in as `j2ee` (password `j2ee`). If you log in as an applicant, such as `romeo` or `juliet`, you will only be able to use the applicant registration functionality. Similarly, customers such as `winston` and `george` will only be able to access job advertisement functionality.

Any attempt to access unauthorized functionality results in a J2EE vendor-specific exception.

Using Roles as the Security Identity

An alternative to propagating the caller's security identity is to define a bean as using a specific role. This is achieved using `deploytool` by selecting the Run As Specified Role option and selecting the appropriate role from the list of roles. The developer or assembler determines whether to use the caller ID or a specific role for a bean.

The `<run-as>` tag is used to define the role for beans that run with a specified role, as shown in Listing 15.4. The role name is defined in the `<role-name>` tag and must specify the name of a role defined in the `<assembly-descriptor>` of the `<ejb-jar>` definition.

LISTING 15.4 Run-As Specified Role in the DD

```
1: <enterprise-beans>
2:   …
3:   <session>
4:     <display-name>AgencyBean</display-name>
5:     <ejb-name>AgencyBean</ejb-name>
```

LISTING 15.4 Continued

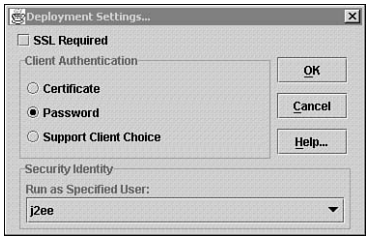
```
 6:     <home>agency.AgencyHome</home>
 7:     <remote>agency.Agency</remote>
 8:     <ejb-class>agency.AgencyBean</ejb-class>
 9:     <session-type>Stateless</session-type>
10:     …
11:     <security-identity>
12:       <run-as>
13:         <description></description>
14:         <role-name>Administrator</role-name>
15:       </run-as>
16:     </security-identity>
17:     …
18:   </session>
19:   …
20: </enterprise-beans>
```

Defining the bean to run using a specific role meets the developer's requirements, but the deployer must also map the role onto a principal in the target security domain for it to be effective. The actual method of defining the principal for the "`Run as role`" security identity is vendor specific. The J2EE RI adds an entry to the `sun-j2ee-ri.xml` descriptor file.

To set the required principal using `deploytool`, click the `Deployment Settings` button and choose the role from the available list in the pop-up window. Figure 15.10 shows the `j2ee` user selected as the principal for a bean that defines the run as role as `Administrator`.

FIGURE 15.10

J2EE RI defining the `<run-as>` security identity.



Note that `deploytool` will only list the users mapped onto the selected role, it will not include the users in groups mapped onto a role.

The Deployment Settings pop-up window is also used to configure client certificates and SSL support used with Web client security.

Defining a bean to run as a specific role is useful when the bean requires special permissions that the client cannot be guaranteed to provide. However, using the "`Run as role`" capability should be used carefully because it effectively negates the authorization process.

Any client with access to the bean automatically gets the appropriate security permissions; therefore, it is imperative that only authorized clients can use the bean.

The "`Run as role`" bean is often a helper Session bean used as an adapter (or wrapper) around other J2EE components, such as Entity beans.

When using the "`Run as role`" security identity, it is usual to define all bean methods as having `<unchecked/>` access allowing access to all roles. If checked access is applied to the methods, the run as role must be allowed; otherwise, the method can never be called.

Programmatic EJB Security

If simple declarative security constraints cannot express all of the security policy rules, the developer must resort to adding security into the EJB code.

The `javax.ejb.EJBContext` interface defines two methods for supporting programmatic security:

- `java.security.Principal getCallerPrincipal()` returns an object defining the principal calling the method. The `Principal` class defines a `getName()` method that returns the name of the principal. The `getCallerPrincipal()` method never returns `null`.
- `boolean isCallerInRole(String roleName)` returns `true` if the caller of the method is in the role passed as a parameter.

These two methods allow the developer to use the client security identity to enable or disable EJB functionality.

Of the two methods, `isCallerInRole()` is considered portable because the developer defines the role name used in the code, and the deployer (or assembler) maps this role reference onto a real role. This allows the developer to write the code without knowing the real role names defined for the application.

The `getCallerPrincipal()` method is considered non-portable because the principal name used is dependent on the authentication mechanism used by the target J2EE server. In practice, as long as principal names are not defined as string literals in the Java code, the `getCallerPrincipal()` method can be used in a portable manner.

Using the Agency case study as an example, you will now implement a real-world solution to authorizing access to your application. Instead of identifying administrators, applicants, and customers, you can simply differentiate between administrators and all other clients.

Some functionality would be restricted to administrators only (creating new skills, for example), but most functionality would be available to all clients.

Client-specific authorization could be implemented so that

- Any client can register as an applicant, but the applicant name must be the same as the client's principal name.
- Any client can register an customer, but the customer name must be the same as the client's principal name.
- Registered applicants or customers can only remove their own details from the system.
- Registered applicants or customers can only log in to access their own data in the system.
- Administrators have unrestricted access to the system.

Listing 15.5 shows the `ejbCreate()` method of the `Advertise` Session bean that has been modified to prevent a client from accessing data that does not match his or her principal name. Administrators are permitted to access data for any client.

LISTING 15.5 The ejbCreate() Method from agency.AdvertiseBean.java

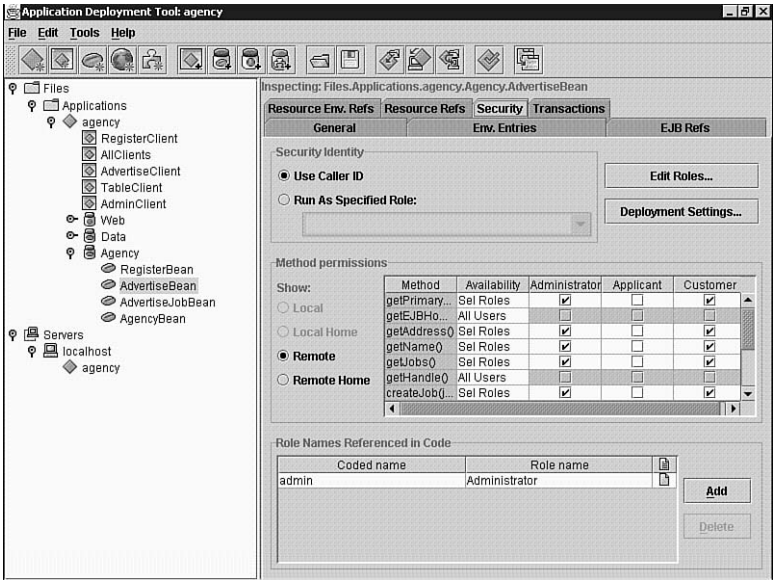
```
 1: public void ejbCreate (String login) throws CreateException {
 2:   try {
 3:     if (ctx.isCallerInRole("admin") ||
 4:       ctx.getCallerPrincipal().getName().equals(login)) {
 5:         customer = customerHome.findByPrimaryKey(login);
 6:     }
 7:     else
 8:       throw new CreateException
➥("Customer name does not match principal name");
 9:   }
10:   catch (FinderException ex) {
11:     error ("Cannot find applicant: "+login,ex);
12:   }
13: }
```

The revised `ejbCreate()` method checks if the caller is not in the `admin` role and rejects the operation if the customer login name does not match the caller's principal name.

The parameter to the `isCallerInRole()` method is a role reference, and this must be mapped onto a real role by the application deployer. Figure 15.11 shows the `Advertise` bean security page in `deploytool` with the role reference defined.

FIGURE 15.11

Defining a role reference.



In Figure 15.11, the coded role of `admin` is mapped onto the real role of `Administrator`. The role references for a bean are defined in the bean's entry in the DD with the `<security-role-ref>` tag, as shown in Listing 15.6.

LISTING 15.6 Role References in the DD

```
<enterprise-beans>
  …
  <session>
    <display-name>AgencyBean</display-name>
    <ejb-name>AgencyBean</ejb-name>
    <home>agency.AgencyHome</home>
    <remote>agency.Agency</remote>
    <ejb-class>agency.AgencyBean</ejb-class>
    <session-type>Stateless</session-type>
    …
    <security-role-ref>
      <role-name>admin</role-name>
      <role-link>Administrator</role-link>
    </security-role-ref>
    …
  </session>
  …
</enterprise-beans>
```

In addition to the change to the `Advertise` EJB, the `Agency` Session bean must also be updated to ensure non-administrator clients can only create or delete customers with a login name equal to the client's principal name (see Listing 15.7).

LISTING 15.7 The Create and Delete Customer Methods from `agency.AgencyBean.java`

```
   1: public void createCustomer(String login, String name, String email)
➥throws DuplicateException, CreateException{
   2:     try {
   3:         if (ctx.isCallerInRole("admin") ||
   4:             ctx.getCallerPrincipal().getName().equals(login)) {
   5:             CustomerLocal customer =
➥customerHome.create(login,name,email);
   6:         }
   7:         else
   8:             throw new IllegalArgumentException(
➥"Cannot create a customer with a different name to the principal name");
   9:     }
  10:     catch (CreateException e) {
  11:         error("Error adding Customer "+login,e);
  12:     }
  13: }
  14:
  15: public void deleteCustomer (String login) throws NotFoundException {
  16:     try {
  17:         if (ctx.isCallerInRole("admin") ||
  18:             ctx.getCallerPrincipal().getName().equals(login)) {
  19:             customerHome.remove(login);
  20:         }
  21:         else
  22:             throw new IllegalArgumentException(
➥"Cannot delete a customer with a different name to the principal name");
  23:     }
  24:     catch (RemoveException e) {
  25:         error("Error removing customer "+login,e);
  26:     }
  27: }
```

The code uses the `java.lang.IllegalArgumentException` to show the error when the customer login name does not match the principal name.

The `admin` to `administrator` role ref must also be added for the `Agency` session EJB.

The `ejbCreate()` method in `agency.AdvertiseJobBean` should also be updated in a similar manner to the `ejbCreate()` method in `agency.AdvertiseBean`, and the `admin` to `administrator` role ref created for the `AdvertiseJob` session EJB.

The revised application can now be deployed and tested.

So far, the security constraints have been added to the Session beans. An alternative approach would have been to add the authorization code to the Entity beans.

The advantage of adding security to the Entity beans is that security is enforced at the data access level, preventing a badly written client from bypassing the security or violating the data integrity. Security is also applied uniformly across all clients and obviates any need for duplicated code to enforce security.

The disadvantage of applying the security within the Entity beans is that it confuses the role of the Entity bean. An Entity bean represents persistent data but should not enforce business rules (other than those required to ensure data integrity). Adding security controls to an Entity bean is adding business rules to the data access layer.

The Agency case study implements the business rules in the Session beans and consistently uses the beans in the GUI client and the Web client. For the Agency case study, the Session bean is the logical place to enforce the security business rules.

Security in Web Applications and Components

The Web security features of J2EE use the same model as the EJB security. Security is implemented using declarations in the deployment descriptor and programming in the Web pages. Authorization is enforced using roles and principals in the same manner as EJB security.

The key concepts for the Web security model are

- *Single login*—A client is only required to authenticate itself once to access all Web pages in the same realm. The Web server defines security realms, and the deployer decides to which realm each Web application belongs. Each realm can use a different authentication mechanism (effectively, a different collection of usernames).
- *Spans multiple applications*—An authenticated client should be able to use Web pages from different Web applications without having to login for each application.
- *Associated with session*—The security credentials must be associated with the servlet session, so that each servlet or JSP can access the credentials when required for programmatic authorization.

The J2EE Web security specifies requirements for client authentication as well as authorization for Web applications.

Web Authentication

The J2EE 1.3 specification does not explicitly specify a Web authentication mechanism, but uses the Servlet 2.3 specification that defines four mechanisms for authenticating users:

- Basic HTTP
- HTTP Digest
- Forms based
- HTTPS Client

The Servlet 2.3 specification can be found on the Sun Microsystems' web site at `http://java.sun.com/products/servlet/download.html`.

Basic HTTP Authentication

The HTTP protocol defines a simple authentication system where the Web server can request the client to supply a username and password. The Web client obtains the username and password of the user and returns them to the Web server for authentication. The popular Web browsers display a simple login form for the user to provide authentication information.

The username and password are returned to the Web server using a simple encoding scheme. Basic HTTP authentication is simple and effective, but it does not provide confidentiality because the username and password are easily obtained by hackers that can monitor network traffic. In reality, hackers will find it almost impossible to monitor network traffic outside of their own organization. However, malicious company employees with the requisite knowledge and software will be able to monitor internal networks.

HTTP Digest Authentication

HTTP Digest authentication works in a similar manner to Basic HTTP authentication except that the username and password are returned in an encrypted form. The encrypted username and password are more secure against illicit monitoring of network traffic.

HTTP digest authentication is not required by the Servlet 2.3 specification because it is not widely supported by Web clients at the present time.

Forms-Based Authentication

J2EE Web applications can specify their own forms-based authentication. This is similar to basic HTTP authentication, but a form is supplied by the application when a user has to be authenticated. The application can supply an authentication form with the same look and feel as the other Web pages in the application, instead of using the simple form provided by a Web browser.

HTTPS Client Authentication

This is the most secure form of authentication because it requires the client to identify itself using a digital certificate. Client authentication is usually implemented by using SSL and is supported by the common Web browsers. This is a large subject area, and there is insufficient space for it to be covered in today's lesson.

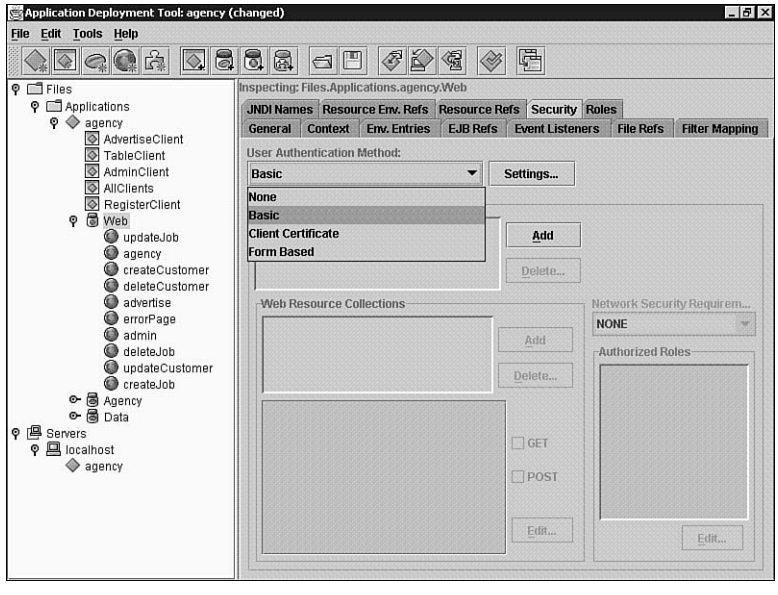
Configuring J2EE RI Basic Authentication

To illustrate the basic features of J2EE Web security, you will need to configure the Web interface to the Agency case study to authenticate users. You will use basic HTTP authentication because it is the simplest mechanism to configure for your Web application.

Start up `deploytool`, open the `agency.ear` file in today's "`examples`" directory and select the Web application you developed on Day 13, "JavaServer Pages" and Day 14 (called `web`). Choose the Security tab and select Basic from the list of options in the User Authentication Method section on the form. Figure 15.12 shows the Basic authentication mechanism being selected from the list of choices.

FIGURE 15.12

Defining Basic HTTP Web authentication.



The Settings button is used to configure additional features for each authentication method. The only configurable property for Basic HTTP authentication is the realm in which to authenticate the user. The J2EE RI only provides one realm, called `default`, for use with Basic HTTP authentication. Although the J2EE RI defines a second realm called `certificate`, but this is for use with HTTPS (SSL)-based client authentication.

Your changes have added a `<login-config>` security constraint to the DD entry for the Web application, as shown in Listing 15.8.

LISTING 15.8 Web Authentication in the DD

```
1: <web-app>
2:   <display-name>Web</display-name>
3:   …
4:   <login-config>
5:     <auth-method>BASIC</auth-method>
6:     <realm-name></realm-name>
7:   </login-config>
8:   …
9: <web-app>
```

Defining the authentication mechanism does not force the client to authenticate itself for pages in the Web application. You will need to add declarative security constraints for the Web pages that have to be protected as discussed in the "Declarative Web Authorization" section later in this chapter. When the user accesses one of the protected pages, the server will use Basic HTTP authentication to obtain the username and password and validate these credentials against the default security realm.

Declarative Web Authorization

Authorized access to Web pages is based on the URL of the Web page. By default, all pages are unprotected, but the DD for a Web application can define security constraints to force a client to authenticate itself before accessing the protected pages.

Authorization is based on roles and constraints. A Web application defines the roles required to access different functionality within the application. One or more constraints can be defined to authorize access to an individual page or a group of pages based on the roles defined for the application.

As with EJB security roles, the deployer must map the role references used in the DD onto principals defined in the target authentication domain.

Configuring J2EE RI Declarative Web Security

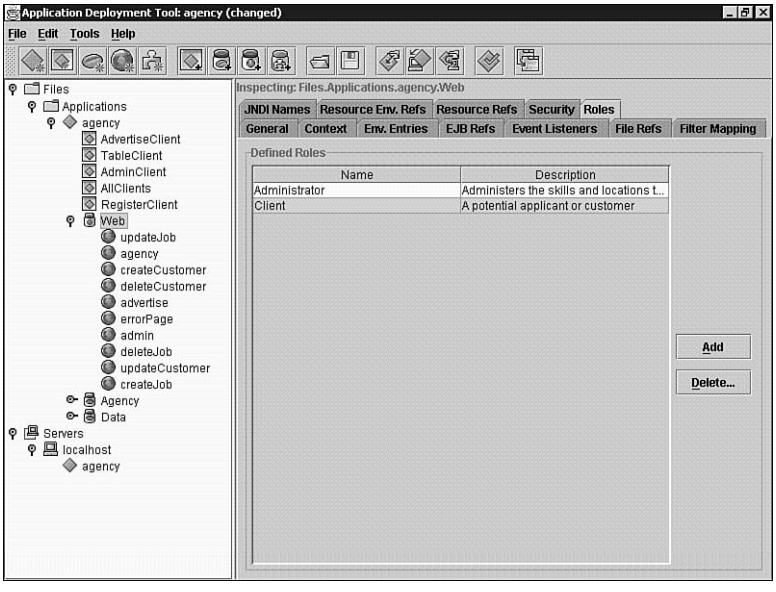
Web application roles are defined in `deploytool` using the Roles page for the Web application. To contrast with the example of EJB security that identified applicants and customers as separate roles, the Web application will treat all potential applicants and customers as clients. You will configure two roles for your Web application, as shown in Table 15.6.

TABLE 15.6 Agency Case Study Roles

Role	*Description*
`Administrator`	Administers the skills and locations tables
`Client`	A potential applicant or customer

Figure 15.13 shows the configured roles for the Agency case study.

FIGURE 15.13
Defining roles for a Web authentication.



After the application roles have been defined, you can add one or more security constraints. A security constraint defines the following:

- A list of roles that are authorized by this constraint
- One or more Web resource collections that define the Web pages protected by this constraint
- A list of protected Web pages each defined by a URL pattern and the HTTP request names `GET` and `POST`

Constraints are applied to the Web application rather than an individual Web component. In `deploytool`, select the Web application (called `web`) and the Security tab (this page is shown in Figure 15.14 after you have made the changes discussed in the rest of this section).

Before adding any constraints, you must decide which parts of the Web application must be protected. The main portal page (`agency.jsp`) should be accessible to all users because this page is used to create new customer or applicant details. Existing customers also use this page to access their data.

Functionality for customers, jobs, and applicants should be protected so that only authenticated clients can use those pages. The administration page (`admin.jsp`) for maintaining skill and location lists should only be accessible to authenticated clients in the `Administration` role.

To enforce these restrictions, you will need two separate constraints for your Web application:

- One constraint to allow clients and administrators to access customer, job, and applicant functionality
- One constraint to allow administrators to maintain the `skill` and `location` lookup tables

On Day 13, you chose aliases for your Web pages to simplify the definition of the security constraints. All of the customer functionality uses aliases of the form `/customer/…`; similarly, applicant functionality is `/applicant/…`. This naming scheme can be used when applying authorization to Web pages.

The following steps will set up the first of these constraints:

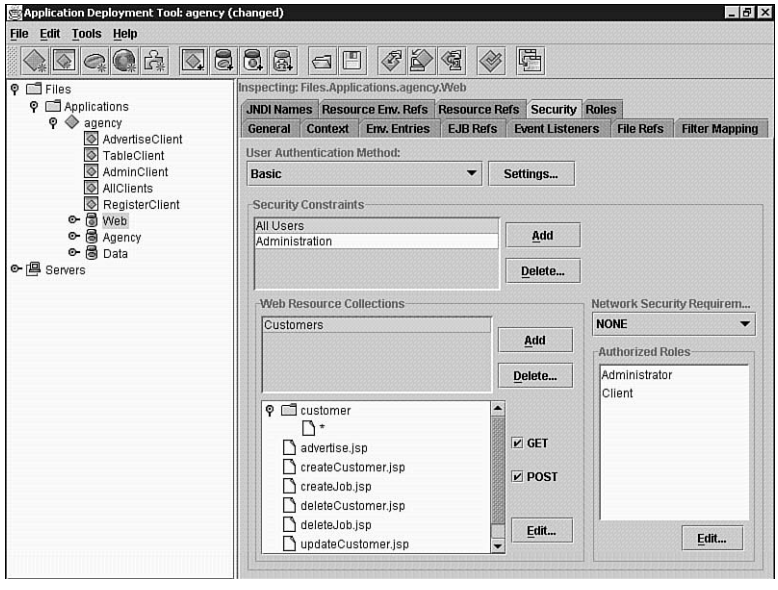
1. In the Security Constraints section, click Add to add a new constraint accept the default name of `SecurityConstraint` because changing it has no effect (the name is not saved in the DD).
2. In the Authorized Roles section, click Edit and add the Client and Administrator roles by using the pop-up screen.
3. In the Web Resource Collections section, click Add to add a new resource bundle and rename this to **`Customers`**.
4. Also in the Web Resource Collections section, click Edit to add the protected URL patterns. You will need to protect the individual page names (in the next step) and the aliases you defined when deploying the application.
5. On the pop-up screen Edit Web Resource Collection page, add the following customer related JSP pages to the collection:
    - `advertise.jsp`
    - `createCustomer.jsp`
    - `createJob.jsp`
    - `deleteCustomer.jsp`

- `deleteJob.jsp`
- `updateCustomer.jsp`
- `updateJob.jsp`

6. On the same page, click the Add URL Pattern button and add the pattern **`/customer/*`**.
7. Click OK to save your changes.
8. Back on the Security page, check the `GET` and `POST` boxes to ensure all HTTP requests to your selected pages are authorized. Your screen will now be similar to the one shown in Figure 15.14.

FIGURE 15.14

Defining Web security constraints.



You have had to protect the real Web page filenames as well as the page aliases because authorization is based on URL patterns and not the physical file location.

You may be wondering why you have protected the actual JSP page when the pages themselves always use the page alias. To answer that, you have to remember that the Web is not a secure environment. There are users who will try to break your security by examining HTTP requests and URLs in an attempt to detect logical naming patterns.

Seeing a URL pattern such as `customer/advertise?customer=winston`, hackers will try different customer names or variations on the Web address to bypass your security.

It only takes one determined hacker to find that the URL `advertise.jsp?customer=winston` works and your security mechanism has been circumvented unless you protect the individual JSP pages as well as the URL pattern `customer/*`.

Where security is concerned, it is better to err on the side of caution and protect everything rather than leave a small loophole for a hacker to exploit.

You must now add a second Resource Collection for protecting access to the applicant registration functionality. Create a new Resource Collection (call it **`Applicants`**) and protect the URL Pattern `/applicant/*` and the following JSP pages:

- `register.jsp`
- `createApplicant.jsp`
- `deleteApplicant.jsp`
- `updateApplicant.jsp`

The last constraint you need is to protect the administrative functionality on the `admin.jsp` page. Because this requires a different set of roles, you will need to

1. Create a new Security Constraint (it will be named `SecurityConstraint1`).
2. Add the `Administration` role to this constraint.
3. Create a new Web Resource Collection and call it **`Administration`**.
4. Add the URL pattern `/admin/*` and the following pages to this collection
    - `admin.jsp`
    - `createLocation.jsp`
    - `createSkill.jsp`
    - `deleteLocation.jsp`
    - `deleteSkill.jsp`
    - `modifyLocation.jsp`
    - `modifySkill.jsp`
    - `updateLocation.jsp`
    - `updateSkill.jsp`

The last portion of the Security screen describes the options for the Network Security Requirements. There are three options available:

- `none` There are no network security requirements.
- `integral` The transfer of the data between server and client must guarantee that the data will not be changed.

- `confidential` The transfer of the data between server and client must guarantee that the data cannot be observed.

In practice, choosing an option other than `none` will usually necessitate the use of SSL. The default value of `none` will suffice during your study of J2EE security.

After you have defined all of these changes, the DD will have the `<security-constraint>` entries, as shown in Listing 15.9.

LISTING 15.9 Web Security Constraints in the DD

```
<web-app>
  <display-name>Web</display-name>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Administration</web-resource-name>
      <url-pattern>/modifySkill.jsp</url-pattern>
      <url-pattern>/createSkill.jsp</url-pattern>
      <url-pattern>/updateLocation.jsp</url-pattern>
      <url-pattern>/admin.jsp</url-pattern>
      <url-pattern>/modifyLocation.jsp</url-pattern>
      <url-pattern>/updateSkill.jsp</url-pattern>
      <url-pattern>/admin/*</url-pattern>
      <url-pattern>/deleteSkill.jsp</url-pattern>
      <url-pattern>/deleteLocation.jsp</url-pattern>
      <url-pattern>/createLocation.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>Administrator</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customers</web-resource-name>
      <url-pattern>/updateCustomer.jsp</url-pattern>
      <url-pattern>/advertise.jsp</url-pattern>
      <url-pattern>/customer/*</url-pattern>
      <url-pattern>/updateJob.jsp</url-pattern>
      <url-pattern>/createJob.jsp</url-pattern>
      <url-pattern>/deleteJob.jsp</url-pattern>
      <url-pattern>/deleteCustomer.jsp</url-pattern>
      <url-pattern>/createCustomer.jsp</url-pattern>
    </web-resource-collection>
    <web-resource-collection>
      <web-resource-name>Applicants</web-resource-name>
```

LISTING 15.9 Continued

```
39:       <url-pattern>/updateApplicant.jsp</url-pattern>
40:       <url-pattern>/applicant/*</url-pattern>
41:       <url-pattern>/register.jsp</url-pattern>
42:       <url-pattern>/createApplicant.jsp</url-pattern>
43:       <url-pattern>/deleteApplicant.jsp</url-pattern>
44:     </web-resource-collection>
45:     <auth-constraint>
46:       <role-name>Administrator</role-name>
47:       <role-name>Client</role-name>
48:     </auth-constraint>
49:     <user-data-constraint>
50:       <transport-guarantee>NONE</transport-guarantee>
51:     </user-data-constraint>
52:   </security-constraint>
53:   ...
54: </web-app>
```

You can now deploy the application with the declarative security enabled. After deploying your application, you can still access the main portal page `http://localhost:8000/agency/agency` without authentication. However, if you select a customer name such as `winston` from the list and get to the advertise page, you will be prompted to enter a username and password. This is the basic HTTP security authentication mechanism.

You must log in with the same name as the customer (`winston`) you selected from the list on the `agency.jsp` page. If you chose any other login name, the EJB programmatic security you added to the `advertise` Session bean will throw an exception because the customer name does not match your principal name.

In the next section, you will use programmatic Web security to remove the need for an existing customer or applicant to select his or her name from a list.

Programmatic Web Authorization

Web applications that are security aware use three methods in the HTTP request object to access the authenticated client's security information.:

- `boolean HttpServletRequest.isUserInRole(String role)` Returns `true` if the client is in the role passed as a parameter.
- `Principal HttpServletRequest.getUserPrincipal()` Returns a `java.security.Principal` object representing the client's principal. Unlike the `EJBContext.getCallerPrincipal()` method, this method can return `null` if the client has not been authenticated.

- `String HttpServletRequest.getRemoteUser()` Returns the principal name of the client or `null` if the client has not been authenticated.

Adding Programmatic Web Security to the Case Study

The Agency case study does not need any programmatic security additions. The component Session beans (`advertise` and `register`) ensure that authenticated clients can only access his or her own details. However, the user interface can be improved by making use of the principal information from the client authentication.

The Agency case study main page (`agency.jsp`) presents the user with a list of customers to select from and a small form for creating a new customer. There is no need for the user to be given a list of customers if an authentication mechanism is used. The user's login name can be used to the appropriate customer `data`. The customer section of the simplified form is shown in Listing 15.10.

LISTING 15.10 Customer Options in `agency.jsp`

```
<H2>Customers</H2>
<FORM action=customer/advertise>
Existing customer: <input type=submit value="Login">
</FORM>
<H3>New Customer</H3>
<FORM action=customer/createCustomer>
<TABLE>
<TR>
  <TD>Login:</TD>
  <TD><INPUT type=text name=login></TD>
</TR>
<TR>
  <TD>Name:</TD>
  <TD><INPUT type=text name=name></TD>
</TR>
<TR>
  <TD>Email:</TD>
  <TD><INPUT type=text name=email></TD>
</TR>
<TR>
  <TD colspan=2><INPUT type=submit value="Create Customer"></TD>
</TR>
</TABLE>
</FORM>
```

An existing customer simply clicks the Login button and the Web authentication form is displayed for the user to login. On successful login, the `advertise.jsp` page is displayed, and this should obtain the current customer from the remote client principal name.

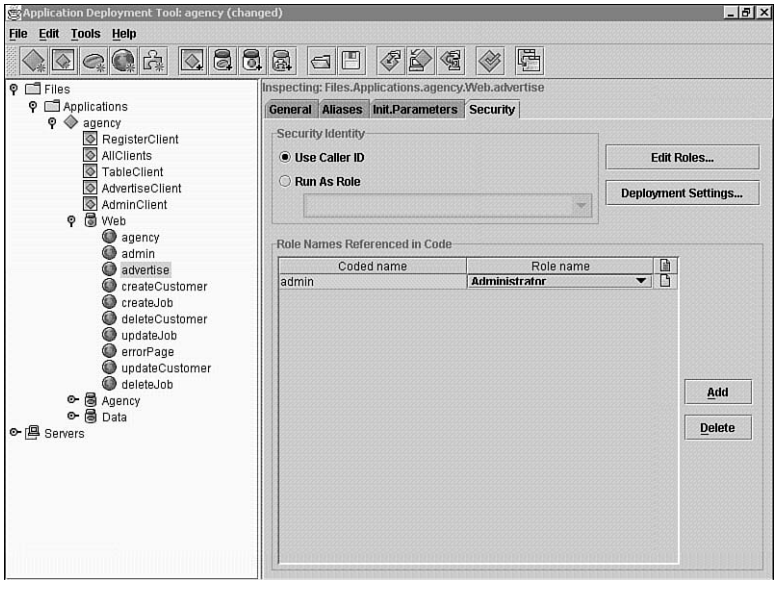
To allow an administrator to login as any user, the form tests the caller's role and, if the authenticated user is an administrator, obtains the customer name from the page parameter. Listing 15.11 shows the new section of code that obtains the customer name.

LISTING 15.11 Customer Name Selection in `advertise.jsp`

```
<%
  String name = null;
  if (request.isUserInRole("admin"))
     name = request.getParameter("customer");
  else
     name = request.getRemoteUser();
%>
<agency:getCust login='<%=name%>'/>
<H2>Customer details for:
➥<jsp:getProperty name="cust" property="login"/></H2>
```

To support the `isUserInRole()` method, an entry must be added to the DD to map the role reference of `admin` onto the real role of `Administrator`. This is done on the Security page of the `advertise` Web application, as shown in Figure 15.15.

FIGURE 15.15
Defining Web role references.



Role references are defined in the `<servlet>` entry in the `<web-app>` section of the DD. Listing 15.12 shows the `admin` role reference added to the DD.

LISTING 15.12 Role Reference in Web Application DD

```
1: <servlet>
2:   <servlet-name>advertise</servlet-name>
3:   <display-name>advertise</display-name>
4:   <jsp-file>/advertise.jsp</jsp-file>
5:   <security-role-ref>
6:     <role-name>admin</role-name>
7:     <role-link>Administrator</role-link>
8:   </security-role-ref>
9: </servlet>
```

In Listing 15.11, if the client is in the `admin` role, the code still uses the request parameter `customer` for the customer name. This supports the requirement for an administrator to be able to modify customer details. The code to do this has been added to the `admin.jsp` file for today's work on the accompanying CD-ROM and is not shown here.

Web applications, like EJBs, can define security roles by using a specific role rather than using the client's ID. The Web application Security page is used to define the available security options.

You can now redeploy the Agency case study and test out the new Web interface. If you test the new changes and login as an applicant (such as `romeo`) rather than as a customer (such as `winston`) your JSP will fail because it cannot create the required `advertise` session EJB. This will prevent any output being returned to your browser, and a browser specific error message will inform you that the requested document has no data. Obviously, in a real application, you would detect the failure to create the Session EJB in the JSP page and report a suitable error to the user.

Using Secure Web Authentication Schemes

The basic HTTP authentication mechanism is suitable for development and testing, but many commercial applications require more "secure" authentication schemes. As discussed earlier, the J2EE Servlet specification requires the server to support HTTPS certificate-based authentication for secure applications by using confidential data.

Because you will never use J2EE RI as a commercial server, learning how to make it secure is not a useful exercise; you have many other features of J2EE to learn in the remaining six days.

If understanding and using HTTPS and SSL is necessary for your understanding of J2EE, this is best done with your commercial Web server. Read the documentation and tutorials with your Web server and give it a go. The basic understanding of security you have gained from today's work will stand you in good stead when implementing certificate-based security.

Security and JNDI

Although JNDI is part of J2SE rather than J2EE, name servers are most commonly used with enterprise applications, many of which use J2EE. The underlying Service Provider implements the security for the naming and directory service. To all intents and purposes, a secure directory service uses LDAP or a service that has an LDAP interface (Active Directory or NDS).

You may want to check back to Day 3, "Naming and Directory Services," to remind yourself about JNDI before reading the rest of this section.

LDAP security is based on three categories:

- `anonymous` No security information is provided
- `simple` The client provides a clear text name and password
- Simple Authentication and Security Layer (SASL) The client and server negotiate an authentication system based on a challenge and response protocol that conforms to RFC2222.

If the client does not supply any security information, the client is treated as an anonymous client.

Security credentials to JNDI are provided as properties. These can be defined in a `jndi.properties` file or supplied as a `HashTable` to the `InitialContext` constructor.

The following JNDI properties provide security information:

- `java.naming.security.authentication` is set to a String to define the authentication mechanism used (one of `none`, `simple`, or a space-separated list of authentication schemes supported by the LDAP server).
- `java.naming.security.principal` is set to the fully-qualified domain name of the client to authenticate.
- `java.naming.security.credentials` is a password or encrypted data (such as a digital certificate) the implementation uses to authenticate the client.

If values for these properties are defined in code using a HashTable, the string constants defined in the javax.naming.Context class should be used instead. These constants are as follows:

- `Context.SECURITY_AUTHENTICATION`
- `Context.SECURITY_PRINCIPAL`
- `Context.SECURITY_CREDENTIALS`

Simple LDAP Authentication

Simple LDAP authentication is easy to use but passes security information, such as the principal name and password, in plain text across the network. Simple authentication is vulnerable to hackers monitoring network data to collect usernames and passwords.

To use simple LDAP authentication, the following properties are needed:

- The authentication is set to simple.
- The security principal is the fully-qualified Distinguished Name (DN) of the LDAP user.
- The security credentials are set to the user's plain text password.

The following example shows how to define simple authentication for a fictitious user called `Winston` with a password of `cigar` (the same user was used in the JNDI examples from Day 3):

```
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, " cn=Winston,ou=Customers,o=Agency,c=us");
env.put(Context.SECURITY_CREDENTIALS, "cigar");

// Create the initial context
DirContext ctx = new InitialDirContext(env);
```

SASL Authentication

If you use strong (`simple` or `anonymous`) authentication, the `java.naming.security.authentication` value consists of a space-separated list of authentication mechanisms. Depending on the LDAP service provider, JNDI can support the following authentication schemes:

- External—Allows JNDI to use any authentication system. The client must define a callback mechanism for JNDI to hook into the client's authentication mechanism.
- GSSAPI (Kerberos v5)—A well-known, token-based security mechanism.
- Digest MD5—Uses the Java Cryptography Extension (JCE) to support client authentication using the MD5 encryption algorithm that has no known decryption technique. This is proposed by RFC2829 to be a mandatory default for LDAP v3 servers.

Additional schemes may also be supported.

An LDAP server stores a list of SASL mechanisms against the attribute `supportedSASLMechanisms` for the root context. Listing 15.13 shows a program that lists out the SASL mechanisms for an LDAP server.

LISTING 15.13 Full Text of `ListSASL.java`

```
 1: import javax.naming.*;
 2: import javax.naming.directory.*;
 3:
 4: public class ListSASL {
 5:     public static void main (String[] args) {
 6:         try {
 7:             // Create initial context
 8:             DirContext ctx = new InitialDirContext();
 9:
10:             // get supported SASL Mechanisms
11:             Attributes attrs =
➥ctx.getAttributes("supportedSASLMechanisms");
12:             NamingEnumeration ae = attrs.getAll();
13:             while (ae.hasMore()) {
14:                 Attribute attr = (Attribute)ae.next();
15:                 System.out.println("  attribute: " + attr.getID());
16:                 NamingEnumeration e = attr.getAll();
17:                 while (e.hasMore())
18:                     System.out.println("    value: " + e.next());
19:             }
20:         }
21:         catch (NamingException ex) {
22:             System.out.println ("Naming error: "+ex);
23:             ex.printStackTrace();
24:             System.exit(1);
25:         }
26:     }
27: }
```

Remember that the default JNDI server for the J2EE RI is a CORBA name server and does not support a directory naming service. You will need to define a `jndi.properties` file in the current directory to define the LDAP server to use. Listing 15.14 shows a suitable file for an LDAP server on 192.168.0.250.

LISTING 15.14 A `jndi.properties` File for an LDAP Server on 192.168.0.250

```
1: java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
2: java.naming.provider.url=ldap://192.168.0.250:389
```

The following code fragment shows how the example user (`Winston`) can define the security credential properties to use Digest MD5:

```
env.put(Context.SECURITY_AUTHENTICATION, "DIGEST-MD5");
env.put(Context.SECURITY_PRINCIPAL, " cn=Winston,ou=Customers,o=Agency,c=us ");
env.put(Context.SECURITY_CREDENTIALS, "cigar");
```

```
// Create the initial context
DirContext ctx = new InitialDirContext(env);
```

To use Digest MD5, the Java Cryptography Extension (JCE) must be installed on your system. JCE is included in JDK 1.4 but must be downloaded from Sun Microsystems' Web site and installed for earlier versions of the JDK.

The subject of JCE and LDAP SASL authentication is a whole day's lesson in its own right, and there isn't time today to do any more work in this area. If you are interested in finding out more about JCE and JNDI security, the JNDI Tutorial on Sun Microsystems' Web site is an excellent starting point.

Summary

Today, you have looked at several aspects of J2EE security. You've studied basic security terminology, including the difference between authentication and authorization.

You have seen how the J2EE specification doesn't specify the authentication schemes that must be used but relies on a server to provide some form of authentication. The authenticated username is known as a J2EE principal.

J2EE authorization is based on roles defined for each EJB JAR or Web JAR in the application. Each authenticated principal can be mapped onto one or more roles.

J2EE uses declarative constraints to define authorization based on the roles defined in the application. Each method in an EJB can be authorized for all principals or a specific list of roles. Similarly, individual Web pages can be authorized for specific roles. This declarative programming de-couples the development of the EJB and Web code from the runtime authentication scheme. Declarative security constraints facilitate the separation of the developer role from the assembler and deployer roles.

Programmatic security is used when simple declarative security cannot express the application's authorization requirements. An EJB or Web page becomes security aware by using methods in the J2EE API to obtain the client's principal name or role. This information can be used to change the behavior of an EJB or Web page based on the client's security credentials.

Adding security to a J2EE application is a simple process. Careful design of the functionality in each EJB or Web page enables an assembler to apply consistent security constraints to several J2EE components comprising a complete application.

Q&A

Q What are six different aspects of security?

A Six aspects of security are

- Authentication
- Authorization
- Confidentiality
- Integrity
- Non-repudiation
- Auditing

Q What are the three participants of the J2EE security domain?

A Principal represents an entity (typically a user) in the authentication system of the target environment. Role represents a security role within the application. Role Reference is used to map a coded role name onto an actual role.

Q What are the two methods of defining J2EE security?

A Declarative and programmatic.

Q What are the two EJB context methods and three HTTP Request methods used in programmatic security?

A
```
EJBContext.isCallerInRole()
EJBContext.getCallerPrincipal()
HttpServletRequest.isUserInRole()
HttpServletRequest.getUserPrincipal()
HttpServletRequest.getRemoteUser()
```

Exercises

Using the code for the `Advertise` EJB as an example, add in security for the `Register` EJB. Use declarative security to restrict access to the `Register` Session bean to members of the J2EE RI `Applicants` group. Add programmatic security to ensure a non-administrator can only create a `Register` EJB using a login name the same as his or her principal name. Update the `Agency` bean to restrict the abilities of a non-administrator to only being able to create and delete applicants with a login name the same as his or her principal name. Don't forget to add the role ref mapping `admin` onto `Administrator` in the `Register` session EJB and the `Register` Web application.

Update the `agency.jsp` and `register.jsp` Web pages to obtain the applicant's name from the security credentials instead of presenting the user with a list of applicant names. Hint: follow the example JSP code for managing customers (`advertise.jsp`) shown in today's lesson.

WEEK 3

DAY 16

Integrating XML with J2EE

Today, we take a bit of departure from J2EE and its emphasis on programming elements to look at what is fast becoming *the lingua franca* of the Internet—the Extensible Markup Language (XML).

Throughout the book so far, you have seen many ways in which XML is used within J2EE applications to describe the structure and layout of the application. Today and tomorrow, you will study XML and its associated APIs and standards to gain a fuller understanding of how XML can be used to exchange data between different components in your applications.

Today you will learn about

- How XML has evolved from the need for platform-independent data exchange
- The relationship between XML and both Standard Generalized Markup Language (SGML) and Hypertext Markup Language (HTML)

- How to create well-formed and valid XML documents
- The Java API for XML Processing (JAXP)
- How to process XML documents with the Simple API for XML (SAX) and the Document Object Model (DOM)
- How XML is used in the J2EE platform

This book is about J2EE, of which XML is just a component. To learn more about XML, take a look at *Sams Teach Yourself XML in 21 Days*, which covers everything you need to know about XML and related standards.

The Drive to Platform-Independent Data Exchange

Applications essentially consist of two parts—functionality described by the code and the data that is manipulated by the code. The in-memory storage and management of data is a key part of any programming language and environment. Within a single application, the programmer is free to decide on how the data is stored and represented. Problems start when the application must exchange data with another application.

One solution is to use an intermediary storage medium, such as a database, and standard tools, such as SQL and JDBC, to gain access to the data in such databases.

But what if the data is to be exchanged directly between two applications, or the applications cannot access the same database. In this case, the data must be encoded in some particular format as it is produced so that its structure and contents can be understood when it is consumed. This has often resulted in the creation of application-specific data formats, such as binary data files (`.dat` files) or text-based configuration files (`.ini`, `.rc`, `.conf`, and so on), in which applications store their information.

Similarly, when exchanging information between applications, purpose-specific formats have arisen to address particular needs. Again, these formats can be text-based, such as HTML for encoding how to display the encapsulated data, or binary, such as those used for sending remote procedure calls. In either case, there tends to be a lack of flexibility in the data representation, causing problems when versions change or when data needs to be exchanged between disparate applications, frequently from different vendors.

XML was developed to address these issues. XML provides a data encoding format that is

- Generic
- Simple
- Flexible
- Extensible
- Portable
- Human readable
- And perhaps most importantly, license-free

Benefits and Characteristics of XML

XML offers a method of putting structured data in a text file. Structured data is data that conforms to a particular format; examples are spreadsheets, address books, configuration parameters, and financial transactions. While being structured, XML is also readable by humans as well as software; this means that you do not need the originating software to access the data.

Origins of XML

XML was created by the World Wide Web Consortium (W3C) who now promote and control the standard. The W3C also promotes and develops a number of other interoperable technologies. The latest XML standard, along with lots of useful information and tools, can be obtained from the WC3 Web site (`www.w3.org`).

XML is a set of rules for designing text formats that describe the structure of your data. XML is not a programming language, so it is therefore easy for non-programmers to learn and use. In devising XML, the originators had a set of design goals which were as follows:

- XML should be straightforward to use over the Internet.
- XML should support a wide variety of applications.
- XML should be compatible with the Standard Generalized Markup Language.
- It must be easy to write programs which process XML documents.
- The number of optional features in XML should be kept to the absolute minimum—ideally, zero.
- XML documents should be human-legible and reasonably clear.
- XML documents should be easy to create.
- Terseness in XML was of minimal importance.

XML is based on the Standard Generalized Markup Language (SGML). SGML is a powerful but complex meta-language that is used to describe languages for electronic document exchange, document management, and document publishing. HTML (probably the best known markup language) is an example of an SGML application. SGML provides a rich and powerful syntax, but its complexity has restricted its widespread use and it is used primarily for technical documentation.

XML was conceived as a means of retaining the power and flexibility of SGML while losing most of its complexity. Although a subset of SGML, XML manages to preserve the best parts of SGML and all of its commonly used features while being more regularly structured and simple to use.

XML is still a young technology but is fast making a significant impact. Already there is an important XML application—XHTML, the successor to HTML. The most popular browsers support XHTML and a number of Web developers are using it and gaining the benefit of a better structured and more flexible language.

Structure and Syntax of XML

In this section, you will explore the syntax of XML and understand what is meant by a well-formed document.

You will often encounter the terms "well formed" and "valid" applied to XML documents. These are not the same. A well-formed document is structurally and syntactically correct, whereas a valid document is also semantically correct. A document can be well-formed but not valid.

The best way to become familiar with the syntax of XML is to write an XML document. To check your XML, you will need access to an XML-aware browser or another XML validator.

An XML browser includes an XML parser. To get the browser to check the syntax and structure of your XML document, simply use the browser to open the XML file. Valid XML will be displayed in a structured way (with indentation). If the XML is not well-formed, an appropriate error message will be given.

An easy way to validate XML is to use a browser. A validating XML parser is available for Microsoft Internet Explorer versions 5 and later. To obtain this, access the Microsoft Developers Network Web site (`msdn.microsoft.com/downloads/samples/internet/xml/xml_validator/`) and follow the download instructions.

Other XML validators are available, such as the Sun Microsystems Multi-Schema XML Validator. This is a Java tool to validate XML documents and can be obtained from `www.sun.com/software/xml/developers/multischema/`.

HTML and XML

At first glance, XML looks very similar to HTML. An XML document consists of elements that have a start and end tag enclosed, just like HTML. In fact, Listing 16.1 is both well-formed HTML and XML.

LISTING 16.1 Example XML and HTML

```
1: <html>
2:   <head><title>Web Page</title></head>
3:   <body>
4:     <H1>Teach Yourself J2EE in 21 Days</H1>
5:     <P>Now you have seen the web page – buy the book</P>
6:   </body>
7: </html>
```

An XML document is only well formed if there are no syntax errors. If you are familiar with HTML, you will be aware that many browsers are lenient with poorly formed HTML documents. Missing end tags and even missing sections will often be ignored and therefore unnoticed until the page is displayed in a more rigorous browser and fails to display correctly.

XML differs from HTML in that a missing end tag will always cause an error.

We will now look at XML syntax so you can understand what is going on.

Structure of an XML Document

The outermost element in an XML document is called the *root* element. Each XML document must have one and only one root element, often called the *top level* element. If there is more than one root element, an error will be generated.

The root element can be preceded by a prolog that contains XML declarations. Comments can be inserted at any point in an XML document. The prolog is optional, but it is good practice to include a prolog with all XML documents giving the XML version being used (all full XML listings in this chapter will include a prolog). A minimal XML document must contain at least one element to be well-formed.

Declarations

There are two types of XML declaration. XML documents may, and should, begin with an *XML declaration*, which specifies the version of XML being used. The following is an example of an XML declaration:

```
<?xml version ="1.0"?>
```

which tells the parser that this document conforms to the XML version 1.0 (WC3 recommendation 10-February-1998). As with all declarations, the XML declaration, if present, should always be placed in the prolog.

The other type of declaration is called an *XML document type declaration* and is used to validate the XML. This will be discussed in more detail in the section titled "Creating Valid XML" later in this chapter.

Elements

An element must have a start tag and an end tag enclosed in < and > characters. The end tag is the same as the start tag except that it is preceded with a / character. The tags are case sensitive, and the names used for the start and end tags must be exactly the same, for example The tags `<Start>...</start>` do not make up an element, whereas `<Start>...</Start>` do (both tags are letter case consistent).

An element name can only contain letters, digits, underscores _, colons :, periods ., and hyphens -. An element name may not begin with a digit, period, or hyphen.

The element may also optionally have attributes and a body. All the elements in Listing 16.2 are well-formed XML elements.

LISTING 16.2 Valid XML Elements

```
1: <start>this is the beginning</start>
2: <date day="16th" Month="February">My Birthday</date>
3: <today yesterday="15th" Month="February"></today>
4: <box color="red"/>
5: <head></head>
6: <end/>
```

Table 16.1 describes each of these elements.

TABLE 16.1 XML Elements

Line in Listing 16.2	*Element Type*	*An Element With*
1	`<tag>text</tag>`	A start tag, body, and end tag
2	`<tag attribute="text">`	Attribute and a `text</tag>`body
3	`<tag attribute="text"> </tag>`	Attribute but no body
4	`<tag attribute="text"/>`	Short form of attribute but no body
5	`<tag></tag>`	A start tag and end tag but no body
6	`<tag/>`	Shorthand for the previous tag

Although the body of an element may contain nearly all the printable Unicode characters, certain characters are not allowed in certain places. To avoid confusion (to human readers as well as parsers) the characters in Table 16.2 should not be used in tag or attribute names. If these characters are required in the body of an element, the appropriate symbolic string in Table 16.2 can be used to represent them.

TABLE 16.2 Special XML Characters

Character	*Name*	*Symbolic Form*
`&`	Ampersand	`&`
`<`	Open angle bracket	`<`
`>`	Close angle bracket	`>`
`'`	Single quotes	`'`
`"`	Double quotes	`"`

The elements in an XML document have a tree-like hierarchy, with elements containing other elements and data. Elements must nest—that is, an end tag must close the textually preceding start tag. This means that

```
<B><I>bold and italic</I></B>
```

is correct, while

```
<B><I>bold and italic</B></I>
```

is not.

Well-Formed XML Documents

An XML document is said to be well-formed if there is exactly one root element, and it and every sub-element has delimiting start and end tags that are properly nested within each other.

The following is a simple XML document with an XML declaration followed by a number of elements. The structure represents a list of jobs that could be used in the Agency case study example. In Listing 16.3, the `<jobSummary>` tag is the root tag followed by a number of jobs.

LISTING 16.3 Example `jobSummary` XML

```
<?xml version ="1.0"?>
<jobSummary>
  <job>
    <customer>winston</customer>
    <reference>Cigar Trimmer</reference>
    <location>London</location>
    <description>Must like to talk and smoke</description>
    <skill>Cigar maker</skill>
    <skill>Critic</skill>
  </job>
  <job>
    <customer>george</customer>
    <reference>Tree pruner</reference>
    <location>Washington</location>
    <description>Must be honest</description>
    <skill>Tree surgeon</skill>
  </job>
</jobSummary>
```

Attributes

Attributes are name/value pairs that are associated with elements. There can be any number of attributes, and an element's attributes all appear inside the start tag. The names of attributes are case sensitive and are limited to certain characters in the same way as those of elements. Attributes can only contain letters, underscores `_`, colons `:`, periods `.`, and hyphens `-`. An attribute name cannot begin with a digit, period, or hyphen.

The value of an attribute is a text string delimited by quotes. Unlike HTML, all attribute values in an XML document must be enclosed in quotes; single or double quotes can be used. Listing 16.4 shows the `jobSummary` XML document re-written to use attributes to hold some of the data.

LISTING 16.4 `JobSummary.xml` XML with Attributes

```
 1: <?xml version ="1.0"?>
 2: <jobSummary>
 3:   <job customer="winston" reference="Cigar Trimmer">
 4:     <location>London</location>
 5:     <description>Must like to talk and smoke</description>
 6:     <skill>Cigar maker</skill>
 7:     <skill>Critic</skill>
 8:   </job>
 9:   <job customer="george" reference="Tree pruner">
10:     <location>Washington</location>
11:     <description>Must be honest</description>
12:     <skill>Tree surgeon</skill>
13:   </job>
14: </jobSummary>
```


This version is preferable to the previous one for two reasons. First, it is easier to check by eye to make sure that every job also has a `customer` and `reference`. Second, in programming terms, the `reference` and the `customer` are items that need to be integrity checked (the `reference` has to be unique and the `customer` must already exist). You generally make an item an attribute when its value is limited in some way. You will see later, in the "Document Type Definitions" section, how to use DTDs and schemas to check the validity of both elements and attributes.

Comments

XML comments have the same syntax as a type of HTML comment. They are introduced by `<!--` and ended with `-->`, for example

```
<!-- this is a comment -->
```

Comments can appear anywhere in a document except within the tags, for example,

```
<item quantity="1 lb">Cream cheese <!-- this is a comment --></item>
```

is acceptable, whereas the following is not

```
<item <!-- this is a comment --> quantity="1 lb">Cream cheese </item>
```

As with commenting code, the comments you add to your XML should be factually correct, useful, and to the point. They should be used to make the XML document easier to read and comprehend.

Any character is allowed in a comment, including those that cannot be used in elements and tags, but to maintain compatibility with SGML, the combination of two hyphens together (`--`) cannot be used within the text of a comment.

Comments should be used to annotate the XML, but you should be aware that the parser might remove the comments, so they may not be always accessible to a receiving application.

Creating Valid XML

As you have seen, XML validators recognize well-formed XML, and this is very useful for picking up syntax errors in your document. Unfortunately, a well-formed, syntactically-correct XML document may still have semantic errors in it. For example, a `job` in Listing 16.4 with no `location` or `skills` does not make sense, but without these elements, the XML document is still well-formed, but not valid.

What is required is a set of rules or constraints that define what is a valid structure for an XML document. There are two common methods for specifying XML rules—the Document Type Definition (DTD) and schemas.

Document Type Definitions

A DTD provides a template that defines the occurrence, and arrangement of elements and attributes in an XML document. Using a DTD, you can define

- Element ordering and hierarchy
- Which attributes are associated with an element
- Default values and enumeration values for attributes
- Any entity references used in the document (internal constants, external files, and parameters)

Entity references are covered in Appendix C, "An Overview of XML," on the CD-ROM.

DTDs originated with SGML and have some disadvantages when compared with XML Schemas, which were developed explicitly for XML. One of these disadvantages is that a DTD is not written in XML, which means you have to learn another syntax to define a DTD.

DTD rules can be included in the XML document as document type declarations, or they can stored in an external document. The syntax is the same in both cases.

If a DTD is being used, the XML document must include a `DOCTYPE` declaration, which is followed by the name of the root element for the XML document. If an external DTD is being used, the declaration also includes the word `SYSTEM` followed by a system identifier (the URI that identifies the location of the DTD file). For example

```
<!DOCTYPE jobSummary SYSTEM "jobSummary.dtd">
```

specifies that the root element for this XML document is `jobSummary` and the remainder of the DTD rules are in the file called `jobSummary.dtd` in the same directory.

An external identifier can also include a public identifier. The public identifier precedes the system identifier and is denoted by the word `PUBLIC`. An XML processor can use the public identifier to try to generate an alternative URI. If the document is unavailable by this method, the system identifier will be used.

```
<!DOCTYPE web-app
➥PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
➥'http://java.sun.com/dtd/web-app_2_3.dtd'>
```

`DOCTYPE`, `SYSTEM` and `PUBLIC` must appear in capitals to be recognized.

Element Type Declarations

The DTD defines every element in the XML document with element type declarations. Each element type declaration takes the following form:

```
<!ELEMENT name ( content ) >
```

For example, for the `jobSummary` XML document in Listing 16.4, the `jobSummary` root element is defined as

```
<!ELEMENT jobSummary ( job* )>
```

The `*` sign indicates that the `jobSummary` element may consist of zero or more job elements. There are other symbols used to designate rules for combining elements and these are listed in Table 16.3.

TABLE 16.3 Occurrence Characters Used in DTD Definitions

Character	*Meaning*
*	Zero or more (not required)
+	One or more (at least one required)

TABLE 16.3 Continued

Character	*Meaning*
?	Element is optional (if present can only appear once)
|	Alternate elements
()	Group of elements

The following defines an XML `job` element that must include one `location`, an optional `description`, and at least one `skill`.

```
<!ELEMENT job (location, description*, skill+)>
```

Defining the Element Content

Elements can contain other elements, or content, or have elements and content. The `jobSummary` element, in Listing 16.4, contains other elements but no text body; whereas the `location` element has a text body but does not contain any elements.

To define an element that has a text body, use the reference `#PCDATA` (Parsed Character DATA). For example, the location element in Listing 16.4 is defined by

```
<!ELEMENT location (#PCDATA)>
```

An element can also have no content (the `<br>` tag in HTML is such an example). This tag would be defined with the `EMPTY` keyword as

```
<!ELEMENT br EMPTY>
```

You will also see elements defined with contents of `ANY`. The `ANY` keyword denotes that the element can contain all possible elements, as well as `PCDATA`. The use of `ANY` should be avoided. If your data is so unstructured that it cannot be defined explicitly, there probably is no point in creating a DTD in the first place.

Defining Attributes

In Listing 16.4, the `job` element has two attributes—`customer` and `reference`. Attributes are defined in an `ATTLIST` that has the following form:

```
<!ATTLIST element attribute type default-value>
```

The *`element`* is the name of the element and *`attribute`* is the name of the attribute. The *`type`* defines the kind of attribute that is expected. A type is either one of the defined constants described in Table 16.4, or it is an enumerated type where the permitted values are given in a bracketed list.

TABLE 16.4 DTD Attribute Types

Type	*Attribute Is a…*
`CDATA`	Character string.
`NMTOKEN`	Valid XML name.
`NMTOKENS`	Multiple XML names.
`ID`	Unique identifier.
`IDREF`	An element found elsewhere in the document. The value for `IDREF` must match the `ID` of another element.
`ENTITY`	External binary data file (such as a gif image).
`ENTITIES`	Multiple external binary files.
`NOTATION`	Helper program.

The `ATTLIST default-value` component defines a value that will be used if one is not supplied. For example

```
<!ATTLIST button visible (true | false) "true").
```

defines that the element `button` has an attribute called `visible` that can be either `true` or `false`. If the attribute is not supplied, because of the default value it will be assumed to be `true`.

The default-value item can also be used to specify that the attribute is `#REQUIRED`, `#FIXED`, or `#IMPLIED`. The meaning of these values is given in Table 16.5.

TABLE 16.5 DTD Attribute Default Values

Default Value	*Meaning*
`#REQUIRED`	Attribute must be provided.
`#FIXED`	Effectively a constant declaration. The attribute must be set to the given value or the XML is not valid.
`#IMPLIED`	The attribute is optional and the processing application is allowed to use any appropriate value if required.

Example DTD

Listing 16.5 is the DTD for the `jobSummary` XML document. Create the DTD in a file called `jobSummary.dtd` in the same directory as your `jobSummary` XML document.

LISTING 16.5 DTD for `jobSummary` XML

```
1: <!ELEMENT jobSummary (job*)>
2: <!ELEMENT job (location, description, skill+)>
3: <!ATTLIST job customer CDATA #REQUIRED>
4: <!ATTLIST job reference CDATA #REQUIRED>
5: <!ELEMENT location (#PCDATA)>
6: <!ELEMENT description (#PCDATA)>
7: <!ELEMENT skill (#PCDATA)>
```

Don't forget to add the following line to the `jobSummary` XML at line 2 (following the PI):

```
<!DOCTYPE jobSummary SYSTEM "jobSummary.dtd">
```

View the `jobSummary.xml` document in your XML browser or other XML validator.

If the browser cannot find the DTD, it will generate an error. Edit `jobSummary.xml`, remove the `customer` attribute, and check that your XML validator generates an appropriate error (such as "Required attribute 'customer' is missing").

Namespaces

When an individual designs an XML structure for some data, he or she is free to choose tag names that are appropriate for the data. Consequently, there is nothing to stop two individuals from using the same tag name for different purposes or in different ways. Consider the job agency that deals with two contract companies, each of which uses a different form of job description (such as those in Listings 16.3 and 16.4). How can an application differentiate between these different types of book descriptions?

The answer is to use namespaces. XML provides namespaces that can be used to impose a hierarchical structure on XML tag names in the same way that Java packages provides a naming hierarchy for Java methods. You can define a unique namespace with which you can qualify your tags to avoid them being confused with those from other XML authors.

An attribute called `xmlns` (XML Name Space) is added to an element tag in a document and is used to define the namespace. For example, line 2 in Listing 16.6 indicates that the tags for the whole of this document are scoped within the `agency` namespace.

LISTING 16.6 XML Document with Namespace

```
1: <?xml version ="1.0"?>
2: <jobSummary xmlns="agency">
3:   <job customer="winston" reference="Cigar Trimmer">
```

LISTING 16.6 Continued

```
 4:     <location>London</location>
 5:     <description>Must like to talk and smoke</description>
 6:     <skill>Cigar maker</skill>
 7:     <skill>Critic</skill>
 8:   </job>
 9:   <job customer="george" reference="Tree pruner">
10:     <location>Washington</location>
11:     <description>Must be honest</description>
12:     <skill>Tree surgeon</skill>
13:   </job>
14: </jobSummary>
```


The `xmlns` attribute can be added to any element in the document to enable scoping of elements, and multiple namespaces can be defined in the same document using a prefix. For example, Listing 16.7 has two namespaces—`ad` and `be`. All the tags have been prefixed with the appropriate namespace and now two different forms of the `job` tag (one with attributes and one without) can coexist in the same file.

LISTING 16.7 XML Document with NameSpaces

```
<?xml version ="1.0"?>
<jobSummary xmlns:ad="ADAgency" xmlns:be="BEAgency">
<ad:job customer="winston" reference="Cigar Trimmer">
    <ad:location>London</ad:location>
    <ad:description>Must like to talk and smoke</ad:description>
    <ad:skill>Cigar maker</ad:skill>
    <ad:skill>Critic</ad:skill>
  </ad:job>
  <be:job>
    <be:customer>george</be:customer>
    <be:reference>Tree pruner</be:refenence>
    <be:location>Washington</be:location>
    <be:description>Must be honest</be:description>
    <be:skill>Tree surgeon</be:skill>
  </be:job>
</jobSummary>
```

Enforcing Document Structure with an XML Schema

As has been already stated, DTDs existed before XML, and they have some limitations:

- A DTD cannot define type information other than characters.
- DTDs were not designed to support namespaces and, although it is possible to add namespaces to a DTD, how to do so is beyond the scope of this book.

- DTDs are not easily extended.
- You can only have one DTD per-document, so you cannot have different definitions of an element in a single document and have them validated with a DTD.
- The syntax for DTDs is not XML. Tools and developers must understand the DTD syntax as well as XML.

To address these issues, a new structure definition mechanism was developed by the W3C to fulfil the role of DTDs while addressing the previously listed limitations. This mechanism is called an XML Schema. It uses XML to represent structure and type information.

The XML Schema standard is split into two parts:

- Specifying the structure and constraints on an XML document
- A way of defining data types, including a set of pre-defined types

Because it is a more powerful and flexible mechanism than DTDs, the syntax for defining an XML schema is slightly more involved. An example of an XML schema for the `jobSummary` XML shown in Listing 16.4 can be seen in Listing 16.8.

The World Wide Web Consortium provides an online XML schema validator. It can be accessed via `www.w3.org/2001/03/webdata/xsv`. If your schema is not accessible via the Web, you will have to upload the file to the W3C site.

LISTING 16.8 XML Schema for Job Agency `JobSummary` XML Document

```
   1: <?xml version="1.0"?>
   2:   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
➥elementFormDefault="qualified">
   3:
   4:     <xsd:element name="jobSummary">
   5:       <xsd:complexType>
   6:         <xsd:sequence>
   7:           <xsd:element name="job" type="jobType" minOccurs="0"
➥maxOccurs="unbounded"/>
   8:         </xsd:sequence>
   9:       </xsd:complexType>
  10:     </xsd:element>
  11:
  12:     <xsd:complexType name="jobType">
  13:       <xsd:sequence>
  14:         <xsd:element name="location" type="xsd:string"/>
  15:         <xsd:element name="description" type="xsd:string"/>
```

LISTING 16.8 Continued

```
 16:         <xsd:element name="skill" type="xsd:string" minOccurs="1"
➥maxOccurs="unbounded"/>
 17:       </xsd:sequence>
 18:         <xsd:attribute name="customer" type="xsd:string" use="required"/>
 19:         <xsd:attribute name="reference" type="xsd:string" use="required"/>
 20:     </xsd:complexType>
 21:   </xsd:schema>
```


The first thing to notice is that this schema exists within a namespace as defined on line 2. The string `xsd` is used by convention for a schema namespace, but any prefix can be used.

Schema Type Definitions and Element and Attribute Declarations

Elements that have sub-elements and/or attributes are defined as *complex types*. In addition to complex types, there are a number of built-in *simple types*. Examples of a few simple types are

- `string` Any combination of characters
- `integer` Whole number
- `float` Floating point number
- `boolean` `true`/`false` or `1`/`0`
- `date` yyyy-mm-dd

A complex type element (one with attributes or sub-elements) has to be defined in the schema and will typically contain a set of element declarations, element references, and attribute declarations. Line 12 of Listing 16.8 is the start of the definition for the `job` tag complex type, which contains three elements (`location`, `description`, and `skill`) and two attributes (`customer` and `reference`).

In a schema, like a DTD, elements can be made optional or required. The `job` element on line 7 is optional because the value of the `minOccurs` attribute is `0`. In general, an element is required to appear when the value of `minOccurs` is `1` or more. Similarly, the maximum number of times an element can appear is determined by the value of `maxOccurs`. This value can be a positive integer or the term `unbounded` to indicate there is no maximum number of occurrences. The default value for both the `minOccurs` and the `maxOccurs` attributes is `1`. If you do not specify the number of occurrences, the element must be present and must only occur once.

Element attributes (like those on lines 18 and 19) can be declared with a `use` attribute to indicate whether the element attribute is `required`, `optional`, or even `prohibited`.

There are more aspects to schemas than it is possible to cover here in this book on J2EE. Visit the WC3 Web site (`www.w3.org`) for more information on XML schemas and all other aspects of XML.

How XML Is Used in J2EE

XML is portable data, and the Java platform is portable code. Add Java APIs for XML that make it easy to use XML and, together, you have the ideal combination:

- Portability of data
- Portability of code
- Ease of use

The J2EE platform bundles all these advantages together.

Enterprises are rapidly discovering the benefits of using J2EE for developing Web Services that use XML for the dissemination and integration of data. Particularly because XML eases both the sharing of legacy data, internally among departments, and the sharing of any data with other enterprises.

J2EE includes the Java API for XML Processing (JAXP) that makes it easy to process XML data with applications written in Java. JAXP embraces the parser standards:

- Simple API for XML Parsing (SAX) for parsing XML as a stream.
- Document Object Model (DOM) to build an in-memory tree representation of an XML document.
- XML Stylesheet Language Transformations (XSLT) to control the presentation of the data and convert it to other XML documents or to other formats, such as HTML. XLST is covered on Day 17, “Transforming XML Documents.”

JAXP also provides namespace support, allowing you to work with multiple XML documents that might otherwise cause naming conflicts.

Internally, J2EE also uses XML to store configuration information about applications. You will have seen the deployment descriptor on many occasions while working through this book.

Parsing XML

So far, you have used Internet Explorer or other third-party tools to parse your XML documents. Now you will look at three APIs that provide a way to access and manipulate the information stored in an XML document so you can build your own XML applications. The Simple API for XML (SAX) defines parsing methods and Document Object

Model (DOM) defines a mechanism for accessing and manipulating well-formed XML. The third is the Java API for XML Processing (JAXP) that you will use to build a simple SAX and DOM parser. The two parsers you will develop effectively echo the input XML structure. Usually, you will want to parse XML to perform some useful function, but simply echoing the XML is a good way to learn the APIs.

JAXP has the benefit that it provides a common interface for creating and using SAX and DOM in Java.

SAX and DOM define different approaches to parsing and handling an XML document. SAX is an event-based API, whereas DOM is tree-based.

With event-based parsers, the parsing events (such as the start and end tags) are reported directly to the application through callback methods. The application implements these callback methods to handle the different components in the document, much like handling events in a graphical user interface (GUI).

Using the DOM API, you will transform the XML document into a tree structure in memory. The application then navigates the tree to parse the document.

Each method has its advantages and disadvantages. Using DOM

- Simplifies the mapping of the structure of the XML.
- Is a good choice when the document is not too large (less than 20Mb). If the document is large, it can place a strain on system resources.
- Most or all of the document needs to be parsed.
- The document is to be altered or written out in a structure that is very different from the original.

Using SAX is a good choice

- If you are searching through an XML document for a small number of tags
- The document is large
- When processing speed is important
- If the document does not need to be written out in a structure that is different from the original

SAX is a public domain API developed cooperatively by the members of the XML-DEV (XML DEVelopment) Internet discussion group.

The DOM is a set of interfaces defined by the W3C DOM Working Group. The latest DOM recommendation can be obtained from the WC3 Web site.

The JAXP Packages

The JAXP APIs are defined in the `javax.xml.parsers` package, which contains two factory classes—`SAXParserFactory` and `DocumentBuilderFactory`.

The packages that define the SAX and DOM APIs are

- `javax.xml.parsers` A common interface for different vendors' SAX and DOM parsers
- `org.w3c.dom` Defines the DOM and all of the components of a DOM
- `org.xml.sax` The SAX API

You will now build two applications—one that uses the SAX API and one that uses DOM.

Parsing XML using SAX

The code examples in this section are written using JAXP 1.1, which supports SAX2.0.

To parse an XML document, you instantiate a `javax.xml.parsers.SAXParseFactory` object to obtain a SAX-based parser. This parser is then used to read the XML document a character at a time.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();

DefaultHandler handler = new XMLParse();
saxParser.parse( new File(argv[0]), handler );
```

Your SAX parser class must extend the public class `org.xml.sax.helpers.DefaultHandler`. This class defines stub methods that receive notification (callbacks) when XML entities are parsed. By default, these methods do nothing, but they can be overridden to do anything you like. For example, a method called `startElement()` is invoked when the start tag for an element is recognized. This method receives the element's name and its attributes. The elements name can be passed in any one of the first three parameters to `startElement()`, see Table 16.6, depending on whether namespaces are being used.

TABLE 16.6 Parameters to the `startElement()` Method

Parameter	*Contents*
`uri`	The namespace URI or the empty string if the element has no namespace URI or if namespace processing is not being performed

TABLE 16.6 Continued

Parameter	*Contents*
`localName`	The element name (without namespace prefix) will be a non-empty string when namespaces processing is being performed
`qualifiedName`	The element name with namespace prefix
attributes	The element's attributes

In the following code example, handling for the qualified name is provided.

```
public void startElement(String uri, String localName, String qualifiedName,
➥Attributes attributes)
                throws SAXException {
    System.out.println ("START ELEMENT " + qualifiedName);
    for (int i = 0; i< attributes.getLength(); i++) {
        System.out.println ("ATTRIBUTE " + attributes.getQName(i) + " = " +
➥attributes.getValue(i));
    }
}
```

This example prints out a statement indicating that a start tag has been parsed followed by a list of the attribute names and values.

A similar `endElement()` method is invoked when an end tag is encountered.

```
public void endElement(String uri, String localName, String qualifiedName)
➥throws SAXException {
    System.out.println ("END ELEMENT " + qualifiedName);
}
```

In the parser in Listing 16.9, not all the XML components will be handled. The default action is for components to be ignored. For a complete list of the other `DefaultHandler` methods, see Table 16.7 or refer to the Java 2 Platform, Enterprise Edition, v 1.3 API Specification.

The parser first checks for the XML document, the name of which is provided on the command line (lines 9–12). After instantiating the `SAXParserFactory` (line 14) and constructing the handler (line 13), the XML file is parsed on line 17—that is all there is to it. Lines 32–57 are where the handler routines are defined. This parser reports the occurrence of the start and end of the document—the start and end of elements and the characters that form the element bodies only.

The complete listing for the SAX Parser is shown in Listing 16.9.

LISTING 16.9 Simple SAX Parser

```
 1: import java.io.*;
 2: import org.xml.sax.*;
 3: import org.xml.sax.helpers.DefaultHandler;
 4: import javax.xml.parsers.*;
 5:
 6: public class XMLParse extends DefaultHandler {
 7:
 8:     public static void main(String argv[]) {
 9:         if (argv.length != 1) {
10:             System.err.println("Usage: XMLParse filename");
11:             System.exit(1);
12:         }
13:         DefaultHandler handler = new XMLParse();
14:         SAXParserFactory factory = SAXParserFactory.newInstance();
15:         try {
16:             SAXParser saxParser = factory.newSAXParser();
17:             saxParser.parse( new File(argv[0]), handler );
18:         }
19:         catch (ParserConfigurationException ex) {
20:             System.err.println ("Failed to create SAX parser:" + ex);
21:         }
22:         catch (SAXException ex) {
23:             System.err.println ("SAX parser exceeption:" + ex);
24:         }
25:         catch (IOException ex) {
26:             System.err.println ("IO exeception:" + ex);
27:         }
28:         catch (IllegalArgumentException ex) {
29:             System.err.println ("Invalid file argument" + ex);
30:         }
31:     }
32:     public void startDocument() throws SAXException {
33:         System.out.println ("START DOCUMENT");
34:     }
35:
36:     public void endDocument() throws SAXException {
37:         System.out.println ("END DOCUMENT");
38:     }
39:
40:     public void startElement(String uri, String localName,
➥String qualifiedName, Attributes attributes)
41:                     throws SAXException {
42:         System.out.println ("START ELEMENT " + qualifiedName);
43:         for (int i = 0; i< attributes.getLength(); i++) {
44:             System.out.println ("ATTRIBUTE " + attributes.getQName(i) +
➥" = " + attributes.getValue(i));
45:         }
46:     }
```

LISTING 16.9 Continued

```
 47:
 48:     public void endElement(String uri, String localName,
➥String qualifiedName) throws SAXException {
 49:         System.out.println ("END ELEMENT " + qualifiedName);
 50:     }
 51:
 52:     public void characters(char[] ch, int start, int length)
➥throws SAXException {
 53:         if (length > 0) {
 54:             String buf = new String (ch, start, length);
 55:             System.out.println ("CONTENT " + buf);
 56:         }
 57:     }
 58: }
```


As already stated, lines 32–57 are the handler callback methods that are called when the corresponding XML entity is parsed. If an entity method is not declared in your parser, the entity is handled by the superclass `DefaultHandler` methods, the default action being to do nothing. Table 16.7 gives a full list of the callback `DefaultHandler` methods that can be implemented.

TABLE 16.7 SAX `DefaultHandler` Methods

Method	*Receives Notification of*
`characters(char[] ch, int start, int length)`	Character data inside an element.
`startDocument()`	Beginning of the document.
`endDocument()`	End of the document.
`startElement(String uri, String localName, String qName, Attributes attributes)`	Start of an element.
`endElement(String uri, String localName, qName)`	End of an element.
`startPrefixMapping (String prefix, String uri)`	Start of a namespace mapping.
`endPrefixMapping (String prefix)`	End of a namespace mapping.
`error(SAXParseException e)`	E recoverable parser error.
`FatalError (SAXParseException e)`	A fatal XML parsing error.

TABLE 16.7 Continued

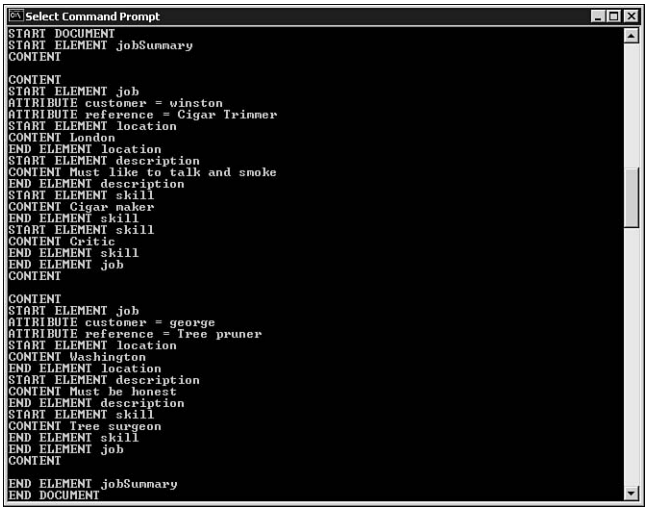
Method	*Receives Notification of*
`Warning (SAXParseException e)`	Parser warning.
`IgnorableWhitespace start, int length).`	Whitespace in the element `(char[] ch, int` contents.
`notationDecl(String name, String publicId, String systemId`	Notation declaration.
`processingInstruction (String target, String data)`	A processing instruction.
`resolveEntity(String publicId, String systemId)`	An external entity.
`skippedEntity(String name)`	A skipped entity (processors may skip entities if they have not seen the declarations (for example, the entity was declared in an external DTD).

This parser can be invoked simply from the command line:

```
> java XMLParse jobSummary.xml
```

The output in Figure 16.1 is produced when this SAX parser is used on the `jobSummary` XML in Listing 16.4.

FIGURE 16.1
SAX parser output.



As you can see, the output is not very beautiful. You might like to improve it by adding indentation to the elements or even getting the output to look like the original XML.

In addition to making this parser more robust, the following functionality could be added:

- Scan element contents for the special characters, such shown in a table, and replacing them with the symbolic strings as appropriate
- Improve the handling of fatal parse errors (`SAXParseException`) with appropriate error messages giving error line numbers
- Use the `DefaultHandler` `error()` and `warning()` methods to handle non-fatal parse errors
- Configure the parser to be namespace aware with `javax.xml.parsers.SAXParserFactory.setNamespaceAware(true)`, so that you can detect tags from multiple sources

You will now build a parser application that uses the DOM API.

Document Object Model (DOM) Parser

When you use the DOM API to parse an XML document, a tree structure representing the XML document is built in memory. You can then analyze the nodes of the tree to discover the XML contents.

The mechanism for instantiating a DOM parser is very similar to that for a SAX parser. A new instance of a `DocumentBuilderFactory` is obtained that is used to create a new `DocumentBuilder`.

The `parse()` method is called on this `DocumentBuilder` object to return an object that conforms to the public `Document` interface. This object represents the XML document tree. The following code fragment creates a DOM parser and reads the XML document from a file called `text.xml`:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse(new File("text.xml");
```

With the `DocumentBuilder.parse()` method, you are not restricted to reading XML only from a file; you can also use a constructed `InputStream` or read from a source defined by a URL.

There are a number of methods provided in the `Document` interface to access the nodes in the tree. These are listed in Table 16.8.

The `normalize()` method should always be used to put all text nodes into a form where there are no adjacent text nodes or empty text nodes. In this form, the DOM view better reflects the XML structure.

As already shown, a DOM parser is instantiated in a similar manner as a SAX parser; the code should be familiar:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse(new File(argv[0]));
```

This is where the similarity ends. At this point, the DOM parser has built an in-memory representation of the document that will look something like Figure 16.2.

The root of the DOM tree is obtained with the `getDocumentElement()` method.

```
Element root = document.getDocumentElement();
```

This method returns an `Element`, which is simply a node that may have attributes associated with it. An element can be the parent to other elements.

There are a number of methods provided in the `Document` interface to access the nodes in the tree, which are listed in Table 16.8. These methods return either a `Node` or a `NodeList` (ordered collection of nodes).

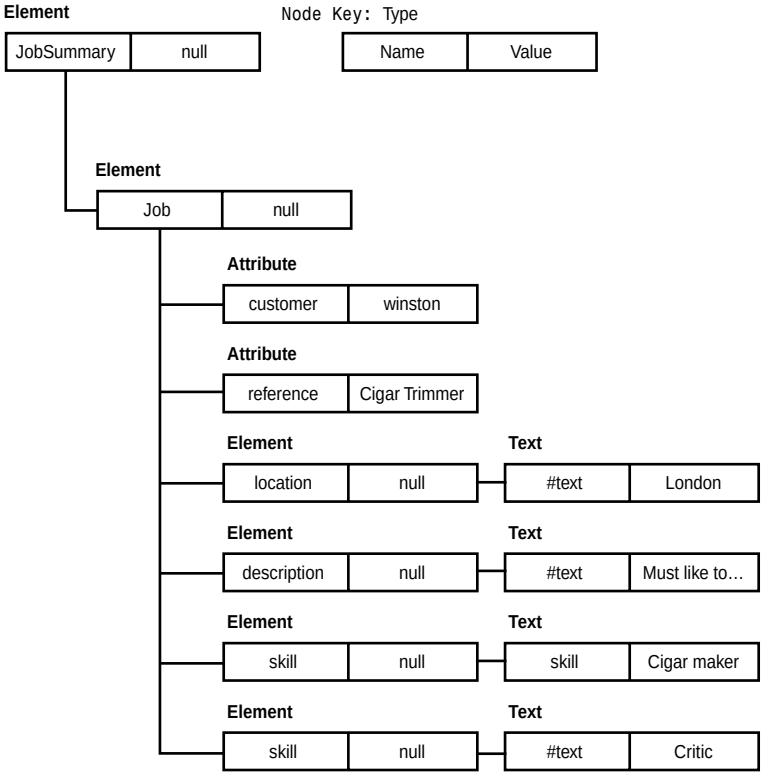
TABLE 16.8 Document Interface Methods to Traverse a DOM Tree

Method Name	*Description*
`getDocumentElement()`	Allows direct access to the root element of the document
`getElementsByTagName(String)`	Returns a `NodeList` of all the elements with the given tag name in the order in which they are encountered in the tree
`getChildNodes()`	A `NodeList` that contains all children of this node

TABLE 16.8 Continued

Method Name	*Description*
`getParentNode()`	The parent of this node
`getFirstChild()`	The first child of this node
`getLastChild()`	The last child of this node
`getPreviousSibling()`	The node immediately preceding this node

FIGURE 16.2
Diagram of DOM tree.



In the DOM application you are about to build, the `getChildNodes()` method is used to recursively traverse the DOM tree. The `NodeList.getLength()` method can then be used to find out the number of nodes in the `NodeList`.

```
NodeList children = node.getChildNodes();
int len = (children != null) ? children.getLength() : 0;
```

In addition to the tree traversal methods, the Node interface provides the following methods to investigate the contents of a node as in Table 16.9.

TABLE 16.9 Document Interface Methods to Inspect DOM Nodes

Method Name	*Description*
`getAttributes()`	A `NamedNodeMap` containing the attributes of a node if it is an `Element` or `null` if it is not.
`getNodeName()`	A string representing name of this node (the tag).
`getNodeType()`	A code representing the type of the underlying object. A node can be one of `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_REFERENCE_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `DOCUMENT_FRAGMENT_NODE`, `NOTATION_NODE`.
`getNodeValue()`	A string representing the value of this node. If the node is a text node, the value will be the contents of the text node; for an attribute node, it will be the string assigned to the attribute. For most node types, there is no value and a call to this method will return `null`.
`getNamespaceURI()`	The namespace URI of this node.
`hasAttributes()`	Returns a `boolean` to indicate whether this node has any attributes.
`hasChildNodes()`	Returns a `boolean` to indicate whether this node has any children.

Listing 16.10 is the full listing of a simple standalone parser that uses DOM. It reads in a file from the command line, builds the parse tree, and outputs elements (including attributes) and text nodes as XML.

LISTING 16.10 Simple DOM Parser

```
1: import javax.xml.parsers.*;
2: import org.xml.sax.*;
3: import java.io.*;
4: import org.w3c.dom.*;
5: import java.util.*;
6:
7: public class DOMParse {
8:
```

LISTING 16.10 Continued

```
 9:     static Document document;
10:
11:     public static void main(String argv[]) {
12:         if (argv.length != 1) {
13:             System.err.println("Usage: DOMParse filename");
14:             System.exit(1);
15:         }
16:         DocumentBuilderFactory factory =
➥DocumentBuilderFactory.newInstance();
17:         try {
18:             DocumentBuilder builder = factory.newDocumentBuilder();
19:             document = builder.parse(new File(argv[0]));
20:             document.getDocumentElement().normalize ();
21:             Element root = document.getDocumentElement();
22:             writeElement(root, "");
23:         }
24:         catch (ParserConfigurationException ex) {
25:             System.err.println ("Failed to create DOM parser:" + ex);
26:         }
27:         catch (SAXException ex) {
28:             System.err.println ("General SAX exeception:" + ex);
29:         }
30:         catch (IOException ex) {
31:             System.err.println ("IO exeception:" + ex);
32:         }
33:         catch (IllegalArgumentException ex) {
34:             System.err.println ("Invalid file argument" + ex);
35:         }
36:     }
37:
38:     private static void writeElement(Node n, String indent) {
39:         StringBuffer name = new StringBuffer(indent);
40:         name.append('<');
       // note where to put / when printing out end tag
41:         int tag_start = name.length();
42:         name.append(n.getNodeName());
43:
44:         NamedNodeMap attrs = n.getAttributes();
45:         int attrCount = (attrs != null) ? attrs.getLength() : 0;
46:         StringBuffer attributes = new StringBuffer();
47:         for (int i = 0; i < attrCount; i++) {
48:             Node attr = attrs.item(i);
49:             attributes.append(' ');
50:             attributes.append(attr.getNodeName());
51:             attributes.append("=\"");
52:             attributes.append(attr.getNodeValue());
53:             attributes.append('"');
54:         }
55:         System.out.print (name);
```

LISTING 16.10 Continued

```
56:         System.out.print (attributes);
57:         System.out.println (">");
58:         name.append('>');
59:
60:         NodeList children = n.getChildNodes();
61:         int len = (children != null) ? children.getLength() : 0;
62:         indent += "  ";
63:         for (int i = 0; i < len; i++) {
64:             Node node = children.item(i);
65:             switch (node.getNodeType())
66:             {
67:               case Node.TEXT_NODE:
68:                 writeText(node, indent);
69:                 break;
70:
71:               case Node.ELEMENT_NODE:
72:                 writeElement(node, indent);
73:                 break;
74:             }
75:         }
76:         name.insert(tag_start, '/');
77:         System.out.println (name);
78:     }
79:
80:     private static void writeText(Node n, String indent) {
81:         String value = n.getNodeValue().trim();
82:         if (value.length() > 0) {
83:             System.out.print(indent);
84:             StringTokenizer XMLTokens =
➥new StringTokenizer(value, "&<>'\"", true);
85:             while (XMLTokens.hasMoreTokens()) {
86:                 String t = XMLTokens.nextToken();
87:                 if (t.length() == 1)  // might be a special char
88:                 {
89:                     if (t.equals("&"))
90:                         System.out.print ("&amp;");
91:                     else if (t.equals("<"))
92:                         System.out.print ("&lt;");
93:                     else if (t.equals(">"))
94:                         System.out.print ("&gt;");
95:                     else if (t.equals("'"))
96:                         System.out.print ("&apos;");
97:                     else if (t.equals("\""))
98:                         System.out.print ("&quot;");
99:                     else
100:                        System.out.print(t);
101:                }
102:                else
103:                    System.out.print(t);
```

LISTING 16.10 Continued

```
104:             }
105:             System.out.println();
106:         }
107:     }
108: }
```


Although at first site this looks more complicated than the SAX parser, most of the additional code is concerned with producing output that conforms to the XML syntax.

Lines 38–57 prints out the start tag with any associated attributes. Lines 59–75 checks for any child nodes and calls the appropriate method according to whether the child node is an XML element or a text node. Line 76 inserts a `/` character into the tag name before printing it out as the end tag.

The `writeText()` method starting on line 80 tokenizes the text contents and replaces the special characters (listed in Table 16.2) with the appropriate XML strings.

Modifying a DOM Tree

We will now look at another use of the DOM API to modify the contents or structure of the XML. Unlike SAX, DOM provides a number of methods that allow nodes to be added, deleted, changed, or replaced in the DOM tree. Table 16.10 summarizes these methods.

TABLE 16.10 Document Interface Methods to Inspect DOM Nodes

Method Name	*Description*
`appendChild(Node newNode)`	Adds the new node to the end of the `NodeList` of children of this node.
`cloneNode(boolean deep)`	Returns a duplicate of a node. The cloned node has no parent. If `deep` is `true`, the whole tree below this node is cloned; if `false`, only the node itself is cloned.
`insertBefore(Node newNode,`	Inserts the `newNode` before `Node refNode)`the existing `refNode`.
`removeChild(Node oldNode)`	Removes the `oldNode` from the list of children.
`replaceChild(Node newNode,`	Replaces the `oldNode` with`Node oldNode)newNode` in the child `NodeList`.

TABLE 16.10 Continued

Method Name	*Description*
`setNodeValue(String nodeValue)`	Set the value of this node, depending on its type; see Table 16.10.
`setPrefix(java.lang.String prefix)`	Set the namespace prefix of this node.

For example, the following code fragment simply creates a new customer element and appends it to the end of the XML document:

```
Node newNode = addXMLNode (document, "Customer", "Columbus");
Element root = document.getDocumentElement();
root.appendChild(newNode);

private static Node addXMLNode (Document document, String name, String text) {
    Element e = document.createElement(name);
    Text t = document.createTextNode(text);
    e.appendChild (t);
    return e;
}
```

The following XML element is added to the XML file that is read in:

```
<customer>Columbus</customer>
```

Java Architecture for XML Binding

DOM is a useful API allowing you to build and transform XML documents in memory. Unfortunately, DOM is somewhat slow and resource hungry. To address these problems, the Java Architecture for XML Binding (JAXB) is being developed through the Java Community Process (JCP) with an expert group consisting of representatives from many commercial organizations.

JAXB provides a mechanism that simplifies the creation and maintenance of XML-enabled Java applications. It does this by using an XML schema compiler (only DTDs at the time of writing) that translates XML DTDs into one or more Java classes, thereby removing the burden from the developer to write complex parsing code.

The generated classes handle all the details of XML parsing and formatting, including code to perform error and validity checking of incoming and outgoing XML documents, which ensures that only valid, error-free XML is accepted.

Because the code has been generated for a specific schema, the generated classes are more efficient than using a generic SAX or DOM parser. Most importantly, a JAXB parser often requires a much smaller footprint in memory than a generic parser.

At the time of writing, JAXB is still in its early release phase.

Differences Between JAXP and JAXB

JAXP and JAXB serve different purposes. Which API you choose depends on the requirements of your application. If you want to transform the data to another format, you should use JAXP, which includes the XSLT transformer API that allows you to transform XML documents, SAX events, or DOM trees (XSLT is covered in Day 17).

JAXP also allows you the flexibility of choosing to validate the data or not. The fact that JAXB requires a DTD guarantees that only valid data is processed.

Classes created with JAXB do not include tree-manipulation capability, which is one factor that contributes to the small memory footprint of a JAXB object tree. If you want to build an object representation of XML data but you need to get around the memory limitations of DOM, you should use JAXB.

These following two bulleted lists summarize the advantages of JAXB and JAXP so that you can decide which one is right for your application.

Use JAXB when you want to

- Access data in memory, but do not need tree manipulation capabilities
- Process only data that is valid
- Convert data to different types
- Generate classes based on a DTD
- Build object representations of XML data

Use JAXP when you want to

- Have flexibility with regard to the way you access the data: either serially with SAX or randomly in memory with DOM
- Use your same processing code with documents based on different DTDs
- Parse documents that are not necessarily valid
- Apply XSLT transformations
- Insert or remove components from an in memory XML tree

Extending the Agency Case Study

As the final part of today's lesson, you will return to code written on Day 10, "Message-Driven Beans," and amend the `Agency` application to handle messages received as XML. First refresh your memory by returning to Day 10 and looking back over the code for the `MessageSender` helper class in Listing 10.4, the `ApplicantMatch` Message-driven bean in Listing 10.5, and the code for the `AgencyBean` and `RegisterBean` Session beans. This code has also been reproduced in the `Examples` directory for Day 16 on the CD-ROM.

The `MessageSender` class will now be modified to create an XML document that contains the applicant's full details to send to the `ApplicantMatch` bean. To achieve this, the following changes to the Day 10 code will be made.

1. Amend the `AgencyBean` and `RegisterBean` Session beans to pass the applicant's full details in the JMS message when an applicant is first registered or the applicant's location or skills are changed.
2. Amend the `MessageSender` helper class to create an XML representation of the data and send this as a message to the `jms/applicantQueue`.
3. Amend the `ApplicantMatch` Message-driven bean to parse the XML message instead of using Entity beans to look up the applicant's details.

The following XML schema represents the applicant's details.

```
<?xml version="1.0"?>
<xsd:schema>
  <xsd:element name="applicantSummary">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="applicant" type="applicantType" minOccurs="0"
➥maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="applicantType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
      <xsd:element name="summary" type="xsd:string"/>
      <xsd:element name="location" type="xsd:string"/>
      <xsd:element name="skill" type="xsd:string" minOccurs="1"
➥maxOccurs="unbounded"/>
    </xsd:sequence>
      <xsd:attribute name="login" type="xsd:string" use="required"/>
      <xsd:attribute name="new" type="xsd:boolean" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

For example the details for `romeo` would be constructed as follows:

```
<?xml version ="1.0"?>
<applicantSummary>
  <applicant login="romeo" new="false">
    <name>Romeo Montague</name>
    <email>romeo@montague.co</email>
    <summary>Dutiful son</summary>
    <location>Wessex</location>
    <skill>Cook</skill>
    <skill>Bodyguard</skill>
  </applicant>
</applicantSummary>
```


Step 1—Change Session Beans

The following changes are required to `AgencyBean`. Replace the `createApplicant()` method with one that calls `messageSender.sendApplicant()` with all the applicants details. The following code shows how this is achieved.

```
public void createApplicant(String login, String name, String email)
➥throws DuplicateException, CreateException{
    try {
        ApplicantLocal applicant = applicantHome.create(login,name,email);
➥messageSender.sendApplicant(applicant.getLogin(), applicant.getName(),
➥applicant.getEmail(), applicant.getSummary(), applicant.getLocation().
➥getName(), applicant.getSkills(), true);
    }
    catch (CreateException e) {
        error("Error creating applicant "+login,e);
    }
    catch (JMSException e) {
        error("Error sending applicant details to message bean "+login,e);
    }
}
```

Similarly, update `RegisterBean` to call the `MessageSender.send()` method with the applicant's full details when a new applicant is added or the details are changed. This time, the information can be obtained from private methods in the `RegisterBean` class.

Replace the `messageSender` method call:

```
messageSender.sendApplicant(getLogin(), getName(), getEmail(), getSummary(),
```

with

```
messageSender.sendApplicant(applicant.getLogin(), applicant.getName(),
➥applicant.getEmail(), applicant.getSummary(), applicant.getLocation().
➥getName(), applicant.getSkills(), false);
```

Note that the final parameter to the `sendApplicant()` method indicates whether this is a new applicant or an updated one. It is set to `true` in `AgencyBean` (to indicate new applicant) and `false` in `RegisterBean` (to indicate that applicant details have been updated).

Step 2—Amend the `MessageSender` Helper Class

The `MessageSender` class now defines some extra methods that construct the XML document. For simplicity, these methods have been left in the `MessageSender` class, but they can be placed in their own helper class if desired.

The following two methods are used to create XML start and an end tags and append the tag to a `StringBuffer` object.

```
private void addStartTag (StringBuffer buf, String tag, String attributes) {
        buf.append("<");
        buf.append(tag);
        buf.append(attributes);
        buf.append(">");
    }
    private void addEndTag (StringBuffer buf, String tag) {
        buf.append("</");
        buf.append(tag);
        buf.append(">");
    }
```

The `addElement()` method constructs an entire XML element:

```
private void addElement (StringBuffer buf, String tag, String attributes,
➥String contents) {
    addStartTag(buf, tag, attributes);
    buf.append (contents);
    addEndTag (buf, tag);
}
```

The applicant XML is created by the `applicantXML()` method:

```
private final String XMLVersion = "<?xml version ='1.0'?>";
private String applicantXML (String applicant, String name,
➥String email, String summary, String location, String[] skills,
➥boolean newApplicant) throws JMSException {
    StringBuffer xmlText = new StringBuffer(XMLVersion);
    addStartTag(xmlText, "applicantSummary", "");
    String newApplicantString = newApplicant?"true'":"false'";
    String attributes = " login='" + applicant + "' new='" + newApplicantString;
    addStartTag(xmlText, "applicant", attributes);
    addElement(xmlText, "name", "", name);
    addElement(xmlText, "email", "", email);
    addElement(xmlText, "summary", "", summary);
    addElement(xmlText, "location", "", location);
    Iterator it = skills.iterator();
```

```
        while (it.hasNext()){
            SkillLocal skill = (SkillLocal) it.next();
            addElement(xmlText, "skill", skill.getName());        }

    addEndTag(xmlText, "applicant");
    addEndTag (xmlText, "applicantSummary");
    return xmlText.toString();
}
```


Finally, the `sendApplicant()` method is changed to construct the XML document before embedding it in the JMS message body:

```
public void sendApplicant(String applicant, String name, String email,
String summary, String location, String[] skills,
➥boolean newApplicant) throws JMSException {
    TextMessage message = queueSession.createTextMessage();
    String xml = applicantXML (applicant, name, email, summary,
➥location, skills, newApplicant);
    message.setText(xml);
    queueSender.send(message);
}
```

Step 3—Amend the `ApplicantMatch` Message-Driven Bean

The `ApplicantMatch` bean now receives all the information required to perform a match between applicants and jobs in the JMS message. Instead of using the `Agency` Entity beans to obtain the location and skill, it parses the XML message. To do this, the `onMessage()` method has been changed to perform the following functions.

1. Get the XML text out of the JMS message.

   ```
   xmlText = ((TextMessage)message).getText();
   ```

2. Build a DOM tree from the XML.

   ```
   DocumentBuilder builder = factory.newDocumentBuilder();
   InputSource is = new InputSource (new StringReader(xmlText));
   document = builder.parse(is);
   document.getDocumentElement().normalize ();
   ```

3. Get the applicant's `login` and `newApplicant` status from the applicant node.

   ```
   String login = getAttribute(document,"applicant", "login");
   String newApplicant = getAttribute (document,"applicant", "new");
       private String getAttribute (Document document, String tag,
   ➥String attribute) {
           NodeList nodes = document.getElementsByTagName(tag);
           NamedNodeMap attrs = nodes.item(0).getAttributes();
           Node n = attrs.getNamedItem(attribute);
           if (n == null)
   ```

```
            return null;
        else
            return n.getNodeValue().trim();
    }
```

4. If this is not a new applicant, make any old matches are removed from the `Match` table.

```
if (newApplicant.equals("false")) {
    matchedHome.deleteByApplicant(login);
}
```

5. Get the applicant's skills and location from the XML elements.

```
String location = getTagContents (document,"location");
Collection appSkills = getMultipleContents (document,"skill");
private String getTagContents (Document document, String tag) {
    NodeList nodes = document.getElementsByTagName(tag);
    NodeList contents = nodes.item(0).getChildNodes();
    int len = (contents != null) ? contents.getLength() : 0;
    Node text = contents.item(0);
    if (len != 1 || text.getNodeType() != Node.TEXT_NODE) {
        return null;
    }
    return text.getNodeValue().trim();
}

private Collection getMultipleContents (Document document, String tag) {
    Collection col = new ArrayList();
    NodeList nodes = document.getElementsByTagName(tag);
    int len = (nodes != null) ? nodes.getLength() : 0;
    for (int i = 0; i < len; i++) {
        NodeList contents = nodes.item(i).getChildNodes();
        Node text = contents.item(0);
        if (text.getNodeType() == Node.TEXT_NODE) {
            col.add(text.getNodeValue().trim());
        }
    }
    return col;
}
```

This replaces the previous code

```
ApplicantLocal applicant = applicantHome.findByPrimaryKey(login);
String location = applicant.getLocation().getName();
Collection skills = applicant.getSkills();
Collection appSkills = new ArrayList();
Iterator appIt = skills.iterator();
while (appIt.hasNext()) {
    SkillLocal as = (SkillLocal)appIt.next();
    appSkills.add(as.getName());
}
```

As before, deploy this code and test by creating new applicants or updating existing applicants. Use the `table.jsp` created on Day 13, "JavaServer Pages,", or any other method you choose, to display the contents of the matched table.

Summary

Today, you have had a very quick and necessarily brief introduction to XML and the APIs and technologies available in J2EE to parse and generate XML data. You have seen how XML can be used to create flexible structured data that is inherently portable. With DTDs and schemas, you appreciated how this data can also be validated. You have been introduced to several different ways of parsing an XML document with SAX, DOM, or JAXB, and you should now recognize the advantages and disadvantages of each technique.

Tomorrow, you will extend your XML knowledge to include XML transformations.

Q&A

Q What are the major characteristics of XML?

A XML is a human readable, structured data-encoding format that is generic, simple, flexible, extensible and free to use.

Q What is the difference between well-formed and valid XML.

A Well-formed XML is syntactically and structurally correct. XML is only valid if it complies with the constraints of a DTD or schema.

Q What are the J2EE APIs and specifications that support the processing of XML?

A The J2EE APIs and specifications that supports XML processing are JAXP Java API for XML Processing, SAX Simple API for XML Parsing, DOM Document Object Model, and XLST for transforming XML documents

Q What are the main differences between SAX and DOM?

A SAX provides a serial event-driven parser. DOM is more flexible in that it builds an in-memory representation of the document that can be manipulated randomly (that is, nodes can be addressed or processed in any order). SAX is generally faster, while DOM can be a heavy user of memory.

Exercises

To practice working with XML, try the following exercise.

1. Extend the Agency case study. Amend the `AdvertiseBean` and `AdvertiseJobBean` Session beans and the `MessageSender` helper class to create an XML document containing all the information about the jobs when they are added or updated. The XML document should have a structure like the following:

```
<?xml version ="1.0"?>
<jobSummary>
  <job customer="winston" reference="Cigar Trimmer" new="false">
    <location>London</location>
    <description>Must like to talk and smoke</description>
    <skill>Cigar maker</skill>
    <skill>Critic</skill>
  </job>
</jobSummary>
```

Update your Message-driven bean called `JobMatch`, written as the exercise on Day 10, that searches through all the applicants to find those suitable to be considered for the job, so that it now takes its information from the XML document. Remember that to be considered for a job, the applicant must match the job's location and at least one skill.

WEEK 3

DAY 17

Transforming XML Documents

In Day 16's lesson, "Integrating XML with J2EE," you studied the basic features of the Extensible Markup Language (XML) and the Java for XML Processing API (JAXP). You can now create XML documents using DOM or Java `OutputStream`/`Writer` objects and process existing XML documents using SAX and DOM. As long as you use XML to store information or transfer information between different components in your application, what you already know about XML is probably sufficient.

But for many applications, you will need to transform your XML documents into other formats, such as HTML, for presentation to a Web client. You may also need to generate a new XML document from an existing one where the new document uses a different XML DTD or schema from the original.

These and other requirements are so ubiquitous among enterprise applications (like those based on J2EE) that tools and standard APIs supporting common capabilities are developing all the time.

In today's work, you will look at

- Techniques for presenting XML data to a Web client
- Applying HTML stylesheets to XML
- The Extensible Stylesheet Language Transformation (XSLT) component of the JAXP for transforming XML documents into other formats
- XALAN, an open-source XSLT implementation from the Apache project
- XSLT compilers, such as `xsltc`

Presenting XML to Clients

XML is a useful way of exchanging data between applications and for moving and storing data within an application. However, XML is not very convenient for presenting data to a user because it primarily describes the content of the data and not how to present it to the user.

Data presented to a user must be formatted so that it is easy to read. XML defines the data and its metadata, but it does not define how to format the data for presentation. In fact, one of the design criteria of XML is *not* to define data presentation formats. Because the same data will be presented in different ways to different users, any attempt to include presentation information would turn the XML document into an incoherent mix of data, metadata, and formatting instructions for multiple output devices.

The technique of presenting XML data to a user involves transforming the data from XML to another format. Typical applications are to transform XML into

- HTML for output to a Web browser
- WML for output to a WAP-enabled mobile phone
- PDF for displaying on any graphic client
- Postscript for output to a printer
- RTF or TeX for presentation to a word processor or text formatter
- XML for presentation to another application that requires the data in a different format

XML can be displayed directly by an XML-aware client using Cascading Stylesheets (CSS) to describe how the XML elements should be displayed.

The transformation of the XML document can be undertaken either by the server that has access to the document or by the client that controls the display. There are pros and cons for each approach. The more work that is done on the server, the fewer clients it can support. However, relying on the client to perform complex transformations tends to rule out the use of many lightweight (thin) clients.

In practice, most current implementations of XSLT perform the transformations on the server. This means that the server must be aware of all possible client display requirements. In practice, this is not a major problem because most clients support a standard presentation language, such as HTML, WML, and PDF, for visual displays and on Postscript for printing.

Presenting XML to Browsers

Web browsers display HTML information. As you saw on Day 16, well written HTML is a well formatted XML- document. Given the close relationship between HTML and XML, it is feasible for an HTML browser to be adapted to present XML. Many recent browsers, such as Mozilla (Netscape 6), Internet Explorer 5, Opera, and countless others, support presentation of XML documents. However, additional formatting information must be provided with the XML data to enable the browser to format the data. By default, XML documents are rendered in plain text, usually with the XML tags highlighted in color and possibly with the tags neatly aligned to highlight the nested tag structure of the document.

Like HTML browsers, there is some variation between the browsers on the market as to how they support presentation of XML. To address this problem, the World Wide Web Consortium (W3C) has defined a standard mechanism called Extensible Stylesheet Language (XSL) for supplying information on transforming an XML document into different formats. The most common use of XSL at the present time is to convert XML into HTML for display by a Web browser, but the technology is much more flexible than this limited use implies.

Early XSL support in popular HTML browsers was based on draft versions of the standard, and some browsers added in their own proprietary extensions. With a fully ratified standard, recent versions of the browsers are now implementing the standard, but at least one manufacturer is maintaining backward compatibility at the expense of standards conformance. You must take into account the variations in client support for XSL when designing your systems. In practice, this means either dictating the client browser and targeting your application to that browser or transforming the data on the server into HTML or another format (such as PDF).

Extensible Stylesheet Language (XSL)

XSL is a family of technologies used to transform XML documents into any other format. The two components of XSL are

- *XSLT*—Extensible Stylesheet Transformations (XSLT) are applied to an XML document to transform it to another format.
- *XSL-FO*—XSL Formatting Objects are used define formatting semantics in a device independent manner.

XSLT has been widely adopted by the computing industry, and you will look in detail at this technology in the "Extensible Stylesheet Transformations" section later in this chapter.

XSL-FO XSL Formatting Objects

Formatting objects define a device-independent grammar for defining how data should be presented. The actual data is intermixed with the formatting objects to define a portable formatted document.

Formatting objects are widely referred to as XSL-FO, but this is not an acronym defined in the W3C standard. You may see XSL-FO written in different forms—XSL:FO, XSLFO, XSL/FO, and others.

Listing 17.1 shows a fragment of an XSL-FO document that defines a table of skills in the Agency case study.

LISTING 17.1 Fragment of an XSL-FO Document

```
 1: <fo:block font-family="sans-serif" color="blue">
 2: <fo:table border-style="solid">
 3:   <fo:table-body>
 4:     <fo:table-row>
 5:       <fo:table-cell padding="1mm">
 6:         <fo:block>Bodyguard</fo:block>
 7:       </fo:table>
 8:       <fo:table-cell padding="1mm">
 9:         <fo:block>Critic</fo:block>
10:       </fo:table>
11:     <fo:table-row>
12:   <fo:table-body>
13: <fo:table>
```

As you can see from listing 17.1, XSL-FO is an XML document defining formatting instructions and data. In this example, the formatting objects provide the bulk of the

document with a very small amount of data. XSL-FO is quite a verbose format and maintains device independence at the cost of some generalizations; the font name on line 1 is `sans-serif` instead of a device-specific font name, such as `Arial`. This helps improve portability of XSL-FO but at the expense of control over the exact rendering of a document on a specific device.

XSL-FO requires two stages for presenting an XML document to a client device.

1. The XML document must be transformed into an XSL-FO document.
2. The XSL-FO document must be transformed into the necessary formatting instructions.

XSLT is used to transform XML into XML-FO. There are several tools around for converting XSL-FO into other formats. One example is the open-source Apache FOP utility that can produce PDF output from an XSL-FO document. FOP can also create a Java Swing application that will display an XSL-FO document using Swing components.

One criticism of XSL-FO is that it does not bring anything new to the process of formatting XML documents. HTML and WML already do a very good job of defining the presentation of data for two of the most popular client devices—PCs and mobile phones. Similarly, PDF defines a portable format for online presentation, and Postscript defines a portable format for printing documents. There is no requirement for another device independent format.

You can find out more about XSL-FO from online Web resources and the book *Sams Teach Yourself XML in 21 Days* from Sams Publishing.

Extensible Stylesheet Transformations (XSLT)

XSLT is a very flexible technique that is used to transform an XML document into a different format, such as XSL-FO, HTML, WML, PDF, or any format you choose, including an XML document conforming to another DTD.

XSLT defines a stylesheet that can be applied to an XML document to transform the XML data into another format. The most common use of XSLT at the moment is to transform XML into HTML for display by a Web browser.

An XSLT stylesheet defines rules that will transform the XML data into the new format. The rules are driven by pattern matching XML elements and attributes in the original XML document. The pattern matching approach enables a single stylesheet to be used with XML documents conforming to different DTDs, provided that there is a reasonable amount of commonality between the XML elements.

A stylesheet transforms any XML document independently from any DTD or Schema to which the document may conform. However, the writer of the stylesheet must be aware of the source document's structure to ensure that the stylesheet achieves the desired result.

An XSLT stylesheet is a well-formed XML document in its own right and conforms to a standard defined by the W3C. Listing 17.2 shows the smallest valid stylesheet called `simple.xsl`; conventionally, stylesheets are stored in files with a `.xsl` suffix.

LISTING 17.2 Full Text of `simple.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

In Listing 17.2, line 2 binds the namespace `xsl` to the namespace URI `http://www.w3.org/1999/XSL/Transform` within the context or the document's root element. Although the example stylesheet apparently does nothing, in fact, it transforms an XML document by removing the XML tags and outputting the remaining text. While this doesn't seem a very useful way of presenting data to a client, it is helpful for understanding the different ways of applying stylesheets to documents.

Applying Stylesheets

There are three ways of applying stylesheets when delivering data to a client such as a Web browser:

- The server uses the stylesheet to transform the XML document into a local file, and the file is sent to the client instead of the original document.
- The stylesheet is sent to the client and then the client transforms the specified XML document for display to the user.
- The server uses the stylesheet to transform the document to a presentation language, such as HTML or WML, and then presents the transformed document to the client.

All three approaches in theory use the same XML data and stylesheet; they differ in where and when the transformation takes place.

Storing Transformed Documents on the Server

Storing the transformed data on the server and presenting this to the client is a good technique when the document changes infrequently and many clients will use the same document. This approach reduces the processing requirements because a new HTML document is only generated when the original data changes.

The downside to this approach is that at least two copies of the data must be retained—the original data and the formatted data. There often may be several copies of the data formatted for each possible client device. Keeping the formatted documents synchronized with original data can become an administrative nightmare. This increased complexity also introduces additional potential for subtle and hard to identify bugs in the application.

Presenting XML Documents and Stylesheets to the Client

Sending the stylesheet and the XML document to the client requires an XML/XSLT-aware client. Given that most clients are Web browsers, this means adding XSLT capabilities to the Web browser.

XSLT support is available in many browsers, such as Netscape 6, Internet Explorer5, Mozilla, Opera, and many others. Sadly, the IE5 and IE6 implementation does not conform to the W3C standard. If you can ensure your customers will use a W3C-conformant browser, presenting XML and a stylesheet to the client is a viable approach. In practice, given the dominance of IE as a Web browser at the present time, portable J2EE applications must adopt a different solution.

Because there are no other widely-used client browsers that support XML transformations (as defined by the W3C), the rest of today's work with XSLT will concentrate on the third means of applying XSLT transformations—that is, on the server.

Transforming the XML Document on the Server

The most common approach to transforming XML for presentation by the client is to do the transformation on the server when the client requests the data. The formatted output is sent to the requesting client. Subsequent requests from the same or different clients must reformat the XML source. This approach ensures that the server retains control of the XML data and only presents the client with the appropriate data. Because the client is just a simple display device, there is no requirement for the client to support XML or XSLT. The drawback is that the server must devote processing time to transforming the data.

There are many tools on the market for supporting XLST, some that are free for personal use and some that require commercial licenses. You will use the XALAN processor from the Apache open-source project for investigating the XSLT transformations discussed in today's chapter. XALAN implements the JAXP framework for including XSLT in your Java programs, and it supports a command-line processing interface for transforming XML documents outside of a programming environment.

Using XALAN with J2EE

You can download XALAN from the following URL on the Apache project Web site.

```
http://xml.apache.org/xalan-j/
```

Download the latest Java version of XALAN to your workstation. Note that XALAN is also provided as a non-Java implementation; make sure you download the Java version. After extracting the XALAN archive, you will have an installation directory named after the version of XALAN (for example, `xalan-j_2_2_D14`).

After installing XALAN, you will need to make the XALAN class files available to the J2EE server and any Java applications you may write. The simplest method of doing this is to install the required XALAN JAR files as JDK extensions.

JDK extensions are installed in the `lib/ext` directory of the Java Runtime Edition (`jre`) directory. Look in the JDK installation directory on your workstation and find the following directory.

Under Windows, it will be

```
%JAVA_HOME%\jre\lib\ext
```

Under Linux/Unix, it will be

```
$JAVA_HOME/jre/lib/ext
```

This is the JDK extension directory. You may also have a separate Java Runtime directory which should also be updated.

Windows users will find the Java Runtime extension directory in `Program Files\JavaSoft\JRE\1.3\lib\ext`.

Linux and Unix users should examine their program search path (`$PATH`) for other Java directories.

You will need to copy the XALAN JAR files into the extension directories for the JDK and any installed Java Runtime Edition to be sure your applications will find the required class files.

In the XALAN installation directory, there is a `bin` sub-directory containing several XALAN files. Copy the following two JAR files from this directory to all the extension directories you have identified on your system:

```
xalan.jar
xml-apis.jar
```

XALAN requires access to an XML parser. The JAXP package from Sun Microsystems contains an XML parser that can be used with XALAN. JAXP is a standard component of J2EE RI 1.3 and JDK 1.4, so it will be available on your workstation if you have installed J2EE RI 1.3.

That is it, XALAN is now ready to use from standalone applications and from within your J2EE applications.

Transforming XML Documents with XALAN

There are two ways of using XALAN:

- From the command line to transform an XML file
- From within a Java program to transform an XML data stream

For introducing XALAN, you will use the simple stylesheet shown in Listing 17.2 and the XML document shown in Listing 17.3.

LISTING 17.3 Full Text of `job.xml`

```
 1: <?xml version ="1.0"?>
 2: <jobSummary>
 3:   <job customer="winston" reference="Cigar Trimmer">
 4:     <location>London</location>
 5:     <description>Must like to talk and smoke</description>
 6:     <!--- skills list for winston -->
 7:     <skill>Cigar maker</skill>
 8:     <skill>Critic</skill>
 9:   </job>
10:   <job customer="george" reference="Tree pruner">
11:     <location>Washington</location>
12:     <description>Must be honest</description>
13:     <!--- skills list for george -->
14:     <skill>Tree surgeon</skill>
15:   </job>
16: </jobSummary>
```

The XML data in Listing 17.3 is similar to the examples you used yesterday when studying XML and JAXP. It defines information about one or more jobs. Each job's primary key is defined using attributes and the job description, location, and skills are defined as child elements. This XML example also contains some XML comments. All aspects of XSLT can be shown using this example.

Using XALAN from the Command Line

Using XALAN from the command line is simply a matter of invoking the XALAN processor class (`org.apache.xalan.xslt.Process`) specifying the source XML file, the stylesheet file, and the output file, as shown below:

```
java org.apache.xalan.xslt.Process -in <input XML> -xsl <stylesheet>
➥-out <output file>
```

If the `-out` parameter is omitted, the output is displayed onscreen. In the Day 17 `Examples` directory on the accompanying CD-ROM, the stylesheets are stored in an `XSL` sub-directory and the XML documents in an `XML` sub-directory. The `run` subdirectory contains a simple batch file and shell script that can be used to run the examples without specifying the full command line. The command line shown in Listing 17.4 could be replaced by run the batch file from the `run` sub-directory by using

```
runXalan job.xml simple.xsl
```

All the examples will use the relative pathnames for the files as stored in the `Examples` directory on the CD-ROM.

Listing 17.4 shows how to use XALAN from the command line to transform the `job.xml` source (Listing 17.3) by using the `simple.xsl` stylesheet (Listing 17.2). The sample files are in the `examples` directory for Day 17 on the accompanying CD-ROM.

LISTING 17.4 Using XALAN from the Command Line

```
1: >java org.apache.xalan.xslt.Process -in XML/job.xml -xsl XSL/simple.xsl
   2: <?xml version="1.0" encoding="UTF-8"?>
   3:
   4:
   5:     London
   6:     Must like to talk and smoke
   7:
   8:     Cigar maker
   9:     Critic
  10:
  11:
  12:     Washington
  13:     Must be honest
  14:
  15:     Tree surgeon
  16:
  17:
```

If you study the output from this command, you will see all the text from the original XML document but none of the processing instructions, XML tags and attributes, or comments. The default transformation rules suppress all but the text elements in the document.

The text elements representing the new lines between elements are retained in the output. This is not a problem when transforming to HTML, but it may be a problem for output formats where newlines are significant. XSL defines tags for handling whitespace as discussed in the "Processing White Space" section in today's lesson.

Using XSLT in Java Applications

The second method of using XSLT is used more often when XML data must be transformed by a Java application. JAXP provides a `javax.xml.transform.Transformer` class that is used to transform XML documents using XSLT.

A `javax.xml.transform.TransformerFactory` is used to create a `Transformer` object. A new `TransformerFactory` object is created by the static `newInstance()` method in the factory class.

The following code creates a new `TransformerFactory`:

```
TransformerFactory factory = TransformerFactory.newInstance();
```

A `Transformer` object is created by a `newTransformer()` method in the factory object and requires a stylesheet as a parameter to the method. The stylesheet must be accessed using a `javax.xml.transform.stream.Source` object. A `StreamSource` object can be constructed from a `java.io.InputStream` (or a `File` or `Reader` object).

The following code constructs an XSLT transformer from the stylesheet file called `simple.xsl`:

```
Source xsl = new StreamSource(new FileInputStream("simple.xsl"));
Transformer transformer = factory.newTransformer(xsl);
```

The `Transformer` method `transform()` is used to transform an XML document using the stylesheet. The `transform()` method takes two parameters—a `Source` defining the XML document and a `javax.xml.transform.stream.Result` for the output file. A `Result` can be constructed from a `java.io.OutputStream` (or a `File` or `Writer` object).

The following lines will transform the XML document `jobs.xml` sending the output to the screen:

```
Source xml = new StreamSource(context.getResourceAsStream("jobs.xml"));
transformer.transform(xml, new StreamResult(System.out));
```

It's as simple as that. Wrapping this code up as a J2EE servlet is a relatively easy operation, as shown by Listing 17.5.

LISTING 17.5 Full Text of `ApplyXSLT.java`

```
 1: import javax.servlet.*;
 2: import javax.servlet.http.*;
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6: import javax.xml.transform.*;
 7: import javax.xml.transform.stream.*;
 8:
 9: public class ApplyXSLT extends HttpServlet {
10:
11:     ServletConfig config;
12:
13:     public void doGet (HttpServletRequest request,
➥HttpServletResponse response)
14:         throws ServletException, IOException,
java.net.MalformedURLException
15:     {
16:         PrintWriter out = response.getWriter();
17:         try {
18:             String source = request.getParameter("source");
19:             int ix = source.lastIndexOf('/');
20:             String xmlDoc = source.substring(0,ix);
21:             String xslDoc = source.substring(ix);
22:             TransformerFactory factory =
➥TransformerFactory.newInstance();
23:
24:             ServletContext context = config.getServletContext();
25:
26:             Source xml =
➥new StreamSource(context.getResourceAsStream(xmlDoc));
27:             Source xsl =
➥new StreamSource(context.getResourceAsStream(xslDoc));
28:
29:             Transformer transformer = factory.newTransformer(xsl);
30:
31:             out.println("<H2>Transformed Document</H2><PRE>");
32:             transformer.transform(xml, new StreamResult(out));
33:             showResource("XSL Stylesheet", xslDoc, out);
34:             showResource("XML Source", xmlDoc, out);
35:             out.print("</PRE>");
36:         }
37:         catch (Exception ex) {
38:             out.println(ex);
39:             ex.printStackTrace(out);
40:         }
41:         out.close();
42:     }
43:
```

LISTING 17.5 Continued

```
  44:     private void showResource(String heading, String name, PrintWriter
out)
➥throws IOException {
  45:         out.println("<H2>"+heading+"</H2>");
  46:         BufferedReader in = new BufferedReader(new InputStreamReader(
➥config.getServletContext().getResourceAsStream(name)));
  47:         String buf;
  48:         while ((buf=in.readLine())!=null) {
  49:             StringTokenizer tok = new StringTokenizer(buf," <>&'\"",true);
  50:             while (tok.hasMoreTokens()) {
  51:                 String s = tok.nextToken();
  52:                 if (s.length()==1)
  53:                 {
  54:                     switch (s.charAt(0)) {
  55:                         case ' ': out.print(" "); break;
  56:                         case '<': out.print("&lt;"); break;
  57:                         case '>': out.print("&gt;"); break;
  58:                         case '&': out.print("&amp;"); break;
  59:                         case '\'': out.print("&apos;"); break;
  60:                         case '"': out.print("&quot;"); break;
  61:                         default: out.print(s); break;
  62:                     }
  63:                 }
  64:                 else
  65:                     out.print(s);
  66:             }
  67:             out.print("<BR>");
  68:         }
  69:         out.print("<P>");
  70:     }
  71:
  72:     public void init(ServletConfig config) throws ServletException {
  73:       super.init(config);
  74:       this.config = config;
  75:     }
  76: }
```

The important lines in the servlet in Listing 17.5 are from 18 to 32. On line 18, the servlet reads a request parameter that defines the XML source file and the stylesheet file, which must be defined in the same Web application as the servlet. The two filenames are passed as a single parameter called `source`, each filename being preceded by a forward slash (`/`).

Before looking at the rest of this servlet, a simple HTML form that lets the user invoke this servlet with the example files provided so far is shown in Listing 17.6.

LISTING 17.6 Full Text of `xsltForm.html`

```
 1: <HTML>
 2: <TITLE>XLST Transformations</TITLE>
 3: <BODY>
 4:   <FORM action=applyXSLT>
 5:     Select an XML document/XSL stylesheet to transform:
 6:     <SELECT name=source>
 7:       <OPTION>/job.xml/simple.xsl
 8:     </SELECT><P>
 9:     <INPUT type=submit>
10:   </FORM>
11: </BODY>
12: </HTML>
```

You will be able to use this form and servlet to examine most of the example transformations shown today.

Returning to Listing 17.5, the code in lines 18 to 21 obtains the names for the XML and stylesheet files from the HTTP `source` parameter.

Lines 26 and 27 use the `getResourceAsStream()` method from the `ServletContext` object to find the named Web application file and return an `InputStream` object that can be used to read the contents of the file. The `getResourceAsStream()` method hides the real location of the file from the servlet enabling the servlet to be deployed as part of any Web application.

In Listing 17.5, lines 22 to 32 perform the actual transformation and write the transformed data to the Web page. In Lines 33 and 34, the `showResource()` method returns the text of the original XML data and stylesheet back to the Web browser for the user to view the original XML document, stylesheet, and transformed output on the single page. The `showResource()` method replaces the reserved HTML characters with their replacement string representation.

To run this demonstration servlet, create a new application called `xslt` and add the following files to the `WAR` file:

- `ApplyXSLT.class`
- `xsltForm.html`
- `jobs.xml`
- `simple.xsl`

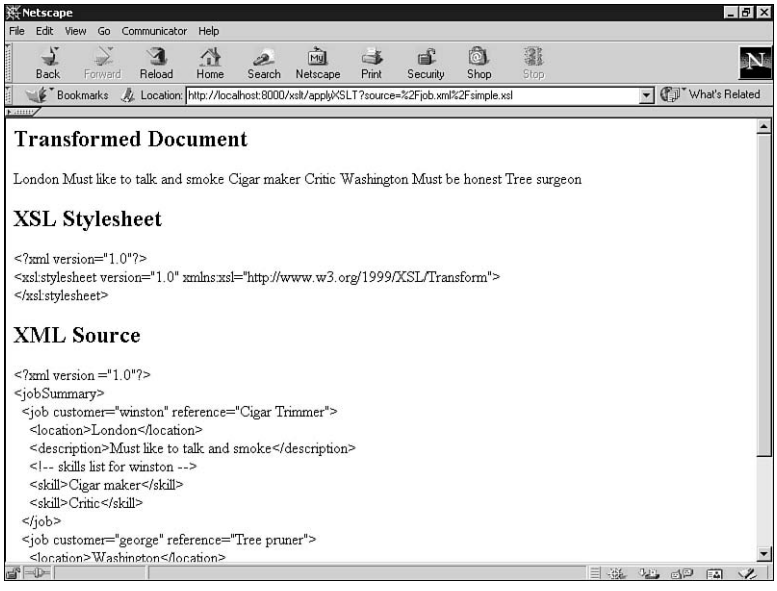
Create a servlet Web application using `ApplyXSLT.class` as the servlet and give it an alias of `/applyXSLT`. Define the Web application context as `/xslt` and deploy the application.

You can access the HTML form using the URL

```
http://localhost:8000/xslt/xsltForm.html
```

Select the XML and stylesheet pair you want to view and click the "Submit" button. Your screen will look similar to the one shown in Figure 17.1.

FIGURE 17.1
Viewing the simple XML transformation.



To look at the later examples in today's lesson, you will need to add XML document and stylesheet entries to the `<select>` list on the HTML form and add the XML document and stylesheet to the `xslt` Web application.

So far, you have seen a very simple XLST transformation. You will now look in more detail at XSLT and its capabilities.

XSLT Stylesheets

In Listing 17.2, you saw a simple stylesheet that used default transformation rules to remove everything except for the text from an XML document. You will now look at how to define your own rules for transforming an XML document.

Rules are based on matching elements in the XML document and transforming the elements into a new document. Text and information from the original XML document can be included or omitted. Components from the XML document are matched using the

XPath notation defined by the W3C. You will learn more about XPath in the "Using XPath with XSLT" section later in today's lesson, after you have looked at some simple XSLT templates.

Template Rules

The most common XSLT template rules are those for matching and transforming elements. The following simple example matches the root node of a document and transforms it into an outline for an HTML document that will be created as you learn more about XSLT's capabilities.

```
<xsl:template match="/">
  <HTML>
    <HEAD> <TITLE>Job Details</TITLE> </HEAD>
    <BODY> </BODY>
  </HTML>
</xsl:template>
```

The `<xsl:template>` defines a new template rule in the stylesheet and its `match` attribute specifies which parts of the XML document will be matched by this rule. The root of a document is matched by the forward slash (`/`); other matching patterns are discussed later in the "Using XPath with XSLT" section.

The body of the `<xsl:template>` element is output in place of the matched element in the original document. In this case, the entire document is replaced by a blank HTML document. No other elements in the document will be matched.

If you want to transform other elements in the original document, you must define additional templates and apply those templates to the body of the matched element. The following text adds an `<xsl:apply-templates/>` element to the rule matching the XML document root:

```
<xsl:template match="/">
  <HTML>
    <HEAD> <title>Job Details</title> </HEAD>
    <BODY> <xsl:apply-templates/> </BODY>
  </HTML>
</xsl:template>
```

When this rule is applied to the transformed root element, the body of the root element is scanned for further template matches. The output from the other rules is inserted at the point where the `<xsl:apply-templates/>` element is defined.

Listing 17.7 shows a simple stylesheet that transforms all of the XML elements into HTML `<STRONG>` elements.

LISTING 17.7 Full Text of `basicHTML.xsl`

```
 1: <?xml version="1.0"?>
 2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 3: <xsl:template match="/">
 4:   <HTML>
 5:     <HEAD> <TITLE>Job Details</TITLE> </HEAD>
 6:     <BODY> <xsl:apply-templates/> </BODY>
 7:   </HTML>
 8: </xsl:template>
 9: <xsl:template match="*">
10:   <P><STRONG><xsl:apply-templates/></STRONG></P>
11: </xsl:template>
12: </xsl:stylesheet>
```


In Listing 17.7, the second rule at lines 9–11 matches every element in the XML document, replaces it with a `<STRONG>` element, and applies all the templates recursively to the body of the XML element.

A stylesheet is an XML document, and you must ensure the XML remains valid when outputting HTML. In Listing 17.7, on line 10, the `<STRONG>` text is enclosed inside an HTML paragraph to ensure that the stylesheet remains valid. Many authors of HTML simply insert the paragraph `<P>` tag at the end of the paragraph. This will not work with stylesheets because the unterminated `<P>` tag is not well-formed XML. Other HTML tags, such as `<BR>` and `<IMG>`, must be treated in a similar manner. There are alternative solutions to the problem of defining HTML documents inside XSLT stylesheets that are outside the scope of this chapter.

Listing 17.8 shows the HTML output from applying the `basicHTML.xsl` stylesheet to the `jobs.xml` file shown in Listing 17.3.

LISTING 17.8 Applying `basicHTML.xsl` to `jobs.xml`

```
1: >java org.apache.xalan.xslt.Process -in XML\job.xml -xsl XSL\basicHTML.xsl
2: <HTML>
3: <HEAD>
4: <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
5: <TITLE>Job Details</TITLE>
6: </HEAD>
7: <BODY>
8: <P>
9: <STRONG>
```

LISTING 17.8 Continued

```
10:
11: <P>
12: <STRONG>
13:
14: <P>
15: <STRONG>London</STRONG>
16: </P>
17:
18: <P>
19: <STRONG>Must like to talk and smoke</STRONG>
20: </P>
21:
22:
23: <P>
24: <STRONG>Cigar maker</STRONG>
25: </P>
26:
27: <P>
28: <STRONG>Critic</STRONG>
29: </P>
30:
31: </STRONG>
32: </P>
33:
34: <P>
35: <STRONG>
36:
37: <P>
38: <STRONG>Washington</STRONG>
39: </P>
40:
41: <P>
42: <STRONG>Must be honest</STRONG>
43: </P>
44:
45:
46: <P>
47: <STRONG>Tree surgeon</STRONG>
48: </P>
49:
50: </STRONG>
51: </P>
52:
53: </STRONG>
54: </P>
55: </BODY>
56: </HTML>
```

In Listing 17.8, the HTML body starts with two STRONG elements corresponding to the <jobSummary> root element and the first <job> element. The nested XML elements <location> and <skill> are output inside <STRONG> tags.

If you studied Listing 17.8 carefully, you will have seen a <META> element inserted into the output at line 4. The XSL processor has identified the output as an HTML document and, on recognizing the HTML <HEAD> element, has inserted the <META> element to identify the contents of the Web page.

The stylesheet must be well formed XML, so any HTML tags must use consistent letter case names for both the start and end tags. HTML is not case sensitive and would allow you to use mismatched names such as <STRONG>.... This example is invalid in XML and will cause the transformation to fail. It is also extremely poor HTML style.

Now that you have seen how the templates are applied to the body of a tag, you might be wondering how not to apply the templates but still output the text of an element. You do this by using the <xsl:value-of select='.'/> tag. This tag outputs the text of the currently selected XML element without applying any more templates either to this element or any of its descendents.

You will use the <xsl:value-of> element when you want to output the text of an XML tag rather than transform it in some way. Listing 17.9 shows a more realistic stylesheet for the jobs.xml example file, and Listing 17.10 shows the transformed document.

LISTING 17.9 Full Text of textHTML.xsl

```
 1: <?xml version="1.0"?>
 2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 3: <xsl:template match="/">
 4:   <HTML>
 5:     <HEAD> <TITLE>Job Details</TITLE> </HEAD>
 6:     <BODY> <xsl:apply-templates/> </BODY>
 7:   </HTML>
 8: </xsl:template>
 9: <xsl:template match="jobSummary">
10:   <H2>Jobs</H2><xsl:apply-templates/>
11: </xsl:template>
12: <xsl:template match="job">
13:   New Job: <P><xsl:apply-templates/></p>
14: </xsl:template>
15: <xsl:template match="description">
```

LISTING 17.9 Continued

```
16:   <P>Descriptiom: <xsl:value-of select="."/></P>
17: </xsl:template>
18: <xsl:template match="location">
19:   <P>Location: <xsl:value-of select="."/></P>
20: </xsl:template>
21: <xsl:template match="skill">
22:   <P>Skill: <xsl:value-of select="."/></P>
23: </xsl:template>
24: </xsl:stylesheet>
```

In Listing 17.9, the leaf elements of `<description>`, `<location>`, and `<skill>` are output as text rather than expanded using the template rules.

Listing 17.9 includes a template for the document root (`match="/"`) and the root element (`match='jobSummary"`). On Day 16, you learned that the document root is the entire XML document, including the processing instructions and comments outside of the root element.

LISTING 17.10 Applying `textHTML.xsl` to `jobs.xml`

```
 1: >java org.apache.xalan.xslt.Process -in XML\job.xml -xsl XSL\textHTML.xsl
 2: <HTML>
 3: <HEAD>
 4: <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
 5: <TITLE>Job Details</TITLE>
 6: </HEAD>
 7: <BODY>
 8: <H2>Jobs</H2>
 9:
10:   New Job: <P>
11:
12: <P>Location: London</P>
13:
14: <P>Descriptiom: Must like to talk and smoke</P>
15:
16:
17: <P>Skill: Cigar maker</P>
18:
19: <P>Skill: Critic</P>
20:
21: </P>
22:
23:   New Job: <P>
24:
25: <P>Location: Washington</P>
```

LISTING 17.10 Continued

```
26:
27: <P>Descriptiom: Must be honest</P>
28:
29:
30: <P>Skill: Tree surgeon</P>
31:
32: </P>
33:
34: </BODY>
35: </HTML>
```

Using the `<xsl:value-of>` tag raises two questions:

- What is the text value of an XML element?
- What does the select attribute do?

These questions are answered in the next two sections.

Text Representation of XML Elements

Every XML node has a textual representation that is used when the `<xsl:value-of>` tag is defined within a template rule. Table 17.1 shows how the textual equivalent of each of the seven XML nodes is obtained.

TABLE 17.1 Text Values of XML Elements

Element Type	*Description*
Document root	The concatenation of all the text in the document
Elements	The concatenation of all the text in the body of the element
Text	The text value of the node, including whitespace
Attributes	The text value of the attribute, including whitespace
Namespaces	The namespace URI that is bound to the namespace prefix associated with the node
Processing Instructions	The text the processing instruction following the target name and including any whitespace
Comments	The text of the comment between the `<!--` and `-->` delimiters

As you can see from Table 17.1, every node has a textual equivalent. The default rules for a stylesheet only include the text values for the document root, elements, and all text nodes. By default, the other four nodes (attributes, namespaces, processing instructions, and comments) are not output. Before you can understand the default rules, you will need to study the XPath notation for matching nodes in an XML document.

Using XPath with XSLT

XPath is means of identifying nodes within an XML document. The W3C identified several aspects of XML that required the ability to identify nodes, for example

- Pointers from one XML document to another called XPointer (the equivalent of `href` in HTML)
- Template rules for XSLT stylesheets
- Schemas

To ensure that the two requirements for identifying nodes share a common syntax, the XPath notation was defined as a separate standard.

An XPath is a set of patterns that can be used to match nodes within an XML document. There are a large number of patterns that can be used to match any part of an XML document. Rather than reproduce the entire XPath specification in today's lesson, you will just study some examples that will help you understand how to use XPath. Further information about XPath can be obtained from the WC3 Web site.

XPath uses the concept of axes and expressions to define a path in the XML document:

- Axes define different parts of the XML document structure.
- Expressions refer to a specific objects within an axis.

Some of the most frequently used axes have special shortcuts to reduce the amount of typing needed. Consider the stylesheet rule you used to match a skill element:

```
<xsl:template match="skill">
  Skill: <xsl:value-of select="."/><P></P>
</xsl:template>
```

This matches a child "`skill"` element using a simple abbreviation. The full XPath notation for this would be:

```
<xsl:template match=" child::skill">
```

The axis is `child` and the expression is an element with the name `skill` (the double colon separates the axis from the expression). The current node that a path is defined from is called the *context node*.

The `child` axis is used to identify all nodes that are immediate children of the context node. Related axes are

- `self` The current node
- `parent` The immediate parent of the context node

- `descendent` Immediate children of the context node, all the children of those nodes, their children, and so on
- `descendent-or-self` All descendent nodes and the current context node
- `ancestor` Any node higher up the node tree that contains context node

There are several other axes defined in the XPath notation.

The `match="."` attribute in the example `<xsl:value-of>` element, shown previously, is another example of a shortcut. The full notation is as follows:

```
Skill: <xsl:value-of select="self::node()"/><P></P>
```

The function `node()` refers to the current context node. Additional functions are

- `name()` The name of the context node instead of the body of the node
- `comment()` Selects a comment node
- `text()` Selects a text node
- `processing-instruction()` Selects a processing instruction node

Some simple XPath expressions are as follows:

- `self::comment()` All comments in the current element
- `child::text()` All the text nodes in the immediate child nodes
- `descendent::node()` All the nodes below the context node
- `descendent-or-self::skill` All the nodes named `skill` below the current node, including the current node

Expressions can be more complex and specify a node hierarchy:

- `job/skill` A skill node that is an immediate child of a job node (in full `child::job/child::skill`)

XPath expressions can be arbitrarily long and can contain the following special expressions:

- `..` The immediate parent node defined as `parent::node()`
- `//` The current node or any descendent as `descendent-or-self::node()`
- `*` Any node in the specified axis
- `|` Used to provide alternate patterns (one pattern or another)

These patterns can be used to identify any node as illustrated by the following examples:

- `jobSummary//skill` Nodes called `skill` defined anywhere below the `jobSummary` node

- `jobSummary/*/skill` `skill` nodes defined as children of children of the `jobSummary` node
- `skill/..` The immediate parent node of a `skill` node
- `location|skill` A `location` or `skill` node
- `parent::comment()|child::text()` Comment nodes in the immediate parent and text nodes in the immediate child
- `/|*` The document root and all elements

Attributes can be selected using the `attribute` axis, which can be abbreviated to `@`. For example,

- `attribute::customer` An the attribute called `customer` of any node (not the node itself)
- `job/@reference` An attribute called `reference` so long as it is associated with a `job` node

In addition to these basic features, XPath supports a powerful matching language supporting variable-like constructs, expressions, and additional functions.

Now that you have a basic understanding of Xpath, you can look at the default rules for a stylesheet.

Default Stylesheet Rules

There are some default stylesheet rules that apply to the whole XML document unless overridden by specific template rules.

The first default rule that ensures all elements are processed is as follows:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

A second rule is used to output the text of text nodes and attributes:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

A third rule suppresses comments and processing instructions:

```
<xsl:template match="processing-instruction()|comment()"/>
```

If an XML element in the source document matches more than one rule, the most specific rule is applied. Consequently, rules defined in a stylesheet will override the default rules.

The second default rule specifies that the text value of attributes should be output, but you can see from Listing 17.4 that the attributes in `job.xml` (Listing 17.3) have not been included. Obviously, there is an extra requirement for processing attributes because this rule has never been invoked.

Processing Attributes

Attributes of XML elements are not processed unless a specific rule is defined to process the element's attributes.

An attribute is processed by using the `<xsl:apply-templates>` rule selecting one or more attributes. The third line in the following rule is the one that applies templates to all attributes:

```
<xsl:template match="*">
  <xsl:apply-templates/>
  <xsl:apply-templates select="@*"/>
</xsl:template>
```


This `<xsl:template>` rule matches all elements and applies templates to the child elements and then that element's attributes. It is the second `<xsl:apply-templates>` rule with the `select="@*"` attribute that ensures that all attributes are output. If you only defined the second `<xsl:apply-templates select="@*">` rule, no output would be produced because the rule had not been applied to elements in the context node.

With this extra information, you can now revisit the `job.xml` file and define a stylesheet that will display the job information in an HTML table. Listing 17.11 shows a stylesheet that will convert a `<job>` element to an HTML table.

LISTING 17.11 Full Text of `table.xsl`

```
   1: <?xml version="1.0"?>
   2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   3: <xsl:template match="job">
   4:   <H2>Job ref: <xsl:value-of select="@customer"/>/
➥<xsl:value-of select="@reference"/></H2>
   5:   <P><xsl:apply-templates select="description"/></P>
   6:   <TABLE border="1">
   7:   <xsl:apply-templates select="skill|location"/>
   8:   </TABLE>
   9: </xsl:template>
  10: <xsl:template match="description">
  11:   <I><xsl:value-of select="."/></I>
  12: </xsl:template>
  13: <xsl:template match="skill|location">
```

LISTING 17.11 Continued

```
  14:   <TR><TD><xsl:value-of select="name()"/>:</TD>
➥<TD><xsl:value-of select="."/></TD></TR>
  15: </xsl:template>
  16: </xsl:stylesheet>
```

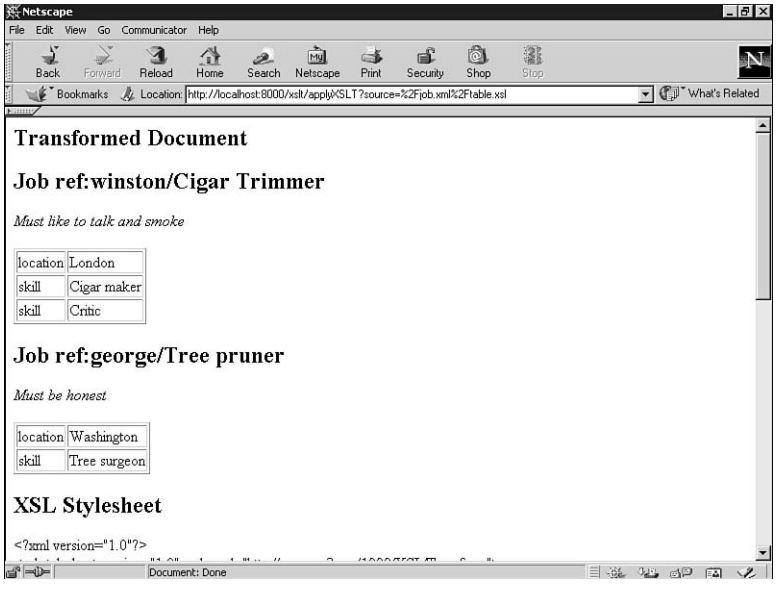
Listing 17.11 brings together several features of stylesheets that have been described previously. The rule at line 3 matches a `<job>` element, and the `customer` and `reference` attributes are inserted into the output at line 4. At line 5, the job `description` child element is output in its own paragraph, and the `skill` and `location` children are output inside an HTML table at lines 6 and 8.

In line 6, the HTML table `border` attribute is enclosed in quotes so that it is valid XML (the same is also true for line 10 and the `colspan` attribute).

Line 12 uses one rule to match the `<location>` or `<skill>` elements. Finally, the name of the selected node is inserted into the output stream using the `name()` function in line 13.

Figure 17.2 shows the result of applying the `table.xsl` stylesheet from Listing 17.10 to the `jobs.xml` file.

FIGURE 17.2
The XML to HTML table transformation.



XSL supports significantly more complex transformation rules than those shown so far. The next section will provide an overview of some of the additional XSL features.

Using Stylesheet Elements

XSL defines about twenty elements for transforming XML documents. So far, you have seen the basic `<xsl:template>` element for defining template rules and the `<xsl:apply-templates>` and `<xsl:value-of>` rules for including data in the output document.

Many transformations can be defined just using these three elements. However, some of the more complex requirements need additional support from XSL.

Processing Whitespace and Text

By default, an XSL transformation retains the whitespace in the original document. This may not be required for the following reasons:

- The whitespace is generally ignored when processing the output.
- Users browsing the transformed document may find the whitespace misleading or annoying.
- Some output document formats may be whitespace sensitive, so the transformation must control the way whitespace is written to the output.

The `<xsl:strip-space>` tag can be used to strip leading and trailing whitespace from an element's body. The tag takes an `elements` attribute that defines from which elements to strip the whitespace. Elements are selected using the XPath notation.

Listing 17.12 shows the `simple.xsl` stylesheet shown in Listing 17.2 enhanced to strip whitespace from all elements.

LISTING 17.12 Full Text of `simpleStrip.xsl`

```
   1: <?xml version="1.0"?>
   2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   3: <xsl:strip-space elements="*"/>
   4: </xsl:stylesheet>
```

Applying this stylesheet to the `jobs.xml` document produces the following output:

```
LondonMust like to talk and smokeCigar makerCriticWashingtonMust
➥be honestTree surgeon
```

No whitespace has been included but, as you can see, this is not particularly readable because the whitespace between the elements has been lost. You could selectively strip whitespace from elements using tags, as shown in the following:

```
<xsl:strip-space elements="jobSummary|job"/>
```

But this will still retain multiple spaces in the non-stripped elements.

Inserting whitespace into the output stream is best done using the <xsl:text> element. Any whitespace inside the <xsl:text> element is retained. You could rewrite the default rule for all elements to include a single blank line before each element, as shown in Listing 17.13.

LISTING 17.13 Full Text of simpleSpace.xsl

```
   1: <?xml version="1.0"?>
   2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   3: <xsl:strip-space elements="*"/>
   4: <xsl:template match="*">
   5:   <xsl:text>
   6: </xsl:text>
   7:   <xsl:apply-templates/>
   8: </xsl:template>
   9: </xsl:stylesheet>
```

The blank line is inserted at lines 5 and 6. Notice how the closing </xsl:text> tag is not indented; otherwise, additional whitespace would have been included. You could not use an empty tag here (<xsl:text/>) because you need to insert the end of line into the output document). The output from applying this stylesheet to jobs.xml is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>


London
Must like to talk and smoke
Cigar maker
Critic

Washington
Must be honest
Tree surgeon
```

Whitespace is automatically stripped from the stylesheet, which is why the indented <xsl:apply-templates/> tag does not insert any whitespace before the output text.

The <xsl:text> tag is also used to insert text containing XML special characters (such as < and &) into the output stream. These characters are defined in the stylesheet using their character reference form (such as <) and are output in the same format. Wrapping the symbols inside an <xsl:text> element with the disable-output-escaping attribute set to yes will output any special symbols as simple Unicode characters. The <xsl:text> element is most often used when the output document is not an XML document, or the output is not well formed XML.

Disabling the output escaping mechanism is dangerous and should only be used when no alternative solution can be found. The `disable-output-escaping` attribute is often abused, much like the `goto` statement in some programming languages.

The following example shows one way of inserting a piece of JavaScript into your Web page (using comments, as described in the "Adding Comments" section later in this chapter, is a better approach):

```
<xsl:template match="SCRIPT">
  <SCRIPT language="javascript">
    <xsl:text disable-output-escaping="yes">
      &lt;!--
        if (n &lt; 0 &amp;&amp;  m &gt; 0) {
          n = m;
        } // --&gt;
    </xsl:text>
  </SCRIPT>
</xsl:template>
```


This transforms to

```
<SCRIPT language="javascript">
<!--
  if (n < 0 && m > 0) {
    n = m;
  } // -->
</SCRIPT>
```

Adding Comments

You can add a comment to a stylesheet as follows:

```
<xsl:template match="job">
  <!--this is a job definition -->
  <xsl:apply-templates/>
</xsl:template>
```

Using XML comments in this way is treated as a stylesheet comment and is not inserted into the output stream. To include a comment in the output document, you must use the `<xsl:comment>` element, as shown in the following:

```
<xsl:template match="job">
  <xsl:comment>this is a job definition</xsl:comment>
  <xsl:apply-templates/>
</xsl:template>
```

The following example shows a better solution to inserting JavaScript in an HTML page:

```
<xsl:template match="SCRIPT">
  <SCRIPT language="javascript">
    <xsl:text>
      <xsl:comment>
        if (n &lt; 0 &amp;amp;amp;  m &gt; 0) {
          n = m;
        } // </xsl:comment>
    </xsl:text>
  </SCRIPT>
</xsl:template>
```

Another advantage of using `<xsl:comment>` is that the actual comment can be derived from data in the source XML document. The following example inserts the job `customer` and `reference` attributes into a comment:

```
<xsl:template match="job">
  <xsl:comment>Job definition for <xsl:value-of select="@customer"/>/
➥<xsl:value-of select="@reference"/></xsl:comment>
  <xsl:apply-templates/>
</xsl:template>
```

Finally, if you want to copy the comments from the original XML document into the transformed document, add the following rule to your stylesheet:

```
<xsl:template match="comment()">
  <xsl:comment><xsl:value-of select="."/></xsl:comment>
</xsl:template>
```

Attribute Values

Some transformations require output element attributes to vary according to the element being processed. As a simple example, think back to Day 16 when you first looked at XML and the simple structure for the job summary, as shown in the following:

```
<?xml version="1.0"?>
<jobSummary>
  <job>
    <customer>winston</customer>
    <reference>Cigar Trimmer</reference>
    <location>London</location>
    <description>Must like to talk and smoke</description>
    <!-- skills list for winston -->
    <skill>Cigar maker</skill>
    <skill>Critic</skill>
  </job>
</jobSummary>
```

This version didn't use attributes to represent the primary key of the job. Imagine that you have to convert this form of document into the new form using attributes. Your first attempt to do this might be as follows:

```
<xsl:template match="job">
  <job
    customer="<xsl:value-of select='./customer'/>"
    job="<xsl:value-of select='./reference'/>"
  />
  <xsl:apply-templates/>
</xsl:template>
```

Unfortunately, this won't work, because XML does not allow you to define elements inside attributes of other elements. You can always insert a < symbol inside an attribute value using `<` but this would not be interpreted as an element.

To get around the XML restriction of not allowing elements to be defined inside attributes, XSLT stylesheets let you insert the value of elements inside attributes by enclosing the XPath name in braces, as shown in the following:

```
<xsl:template match="job">
  <job customer="{./customer}" job="{./reference}" />
  <xsl:apply-templates select='location|description|skill'/>
</xsl:template>
```

To prevent the nested `<customer>` and `<reference>` elements from being output by the default template rules, the `<apply-templates>` element selects the required child elements of the job element. An alternative syntax that excludes the unwanted elements and doesn't need you to explicitly list the required elements uses an XPATH expression notation that has not been discussed today. For completeness, this rule it is as follows:

```
<xsl:apply-templates select="./*[name()!='customer' and name()!='reference']"/>
```

If you want to convert the new style `job` element (with attributes) back to the one with nested elements, you use the following rule:

```
<xsl:template match="job">
  <job>
    <customer><xsl:value-of select="@customer/></customer>
    <reference><xsl:value-of select="@reference/></reference>
    <xsl:apply-templates/>
  </job>
</xsl:template>
```

Creating and Copying Elements

In the previous section, you learned how to convert one XML document to another by converting nested elements into tags. But what do you do if you want to convert an attribute into an element where the attribute value is the name of the element, or vice versa?

As a simple example, consider a Deployment Descriptor (DD) for two Session beans, as shown in Listing 17.14.

LISTING 17.14 Full Text of `dd.xml`

```
   1: <?xml version="1.0" encoding="UTF-8"?>
   2: <!-- DOCTYPE ejb-jar PUBLIC
➥'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
➥'http://java.sun.com/dtd/ejb-jar_2_0.dtd' -->
   3: <ejb-jar>
   4:   <display-name>Agency</display-name>
   5:   <enterprise-beans>
   6:     <session>
   7:       <display-name>AgencyBean</display-name>
   8:       <ejb-name>AgencyBean</ejb-name>
   9:       <home>agency.AgencyHome</home>
  10:       <remote>agency.Agency</remote>
  11:       <ejb-class>agency.AgencyBean</ejb-class>
  12:       <session-type>Stateless</session-type>
  13:       <transaction-type>Bean</transaction-type>
  14:     </session>
  15:     <session>
  16:       <display-name>AdvertiseBean</display-name>
  17:       <ejb-name>AdvertiseBean</ejb-name>
  18:       <home>agency.AdvertiseHome</home>
  19:       <remote>agency.Advertise</remote>
  20:       <ejb-class>agency.AdvertiseBean</ejb-class>
  21:       <session-type>Stateful</session-type>
  22:       <transaction-type>Bean</transaction-type>
  23:     </session>
  24:   </enterprise-beans>
  25: </ejb-jar>
```

Imagine that a different application (or a future version of J2EE) decided that stateless and stateful Session beans were sufficiently different to warrant using different elements. For example,

```
<Stateless>
  <ejb-name>AgencyBean</ejb-name>
  …
</Stateless>
<Stateful>
  <ejb-name>AdvertiseBean</ejb-name>
  …
</Stateful>
```

To make this transformation, you need a rule that can generate an element whose name is derived from the original XML document. The XSL element that does this is called `<xsl:element>`, and the transformation shown previously is achieved by the following rule:

```
<xsl:template match="session">
  <xsl:element name="{./session-type}">
    <xsl:apply-templates/>
  </xsl:element>
```

```
</xsl:template>
<xsl:template match="session/session-type"/>
```

The `name` attribute to the `<xsl:element>` is the name of the element to define. In this example, the name is the value of the `session-type` child element (remember that braces take the value of a node when defining attributes).

The second rule in the previous example (`<xsl:element name="{./session-type}">`) ensures that the `session-type` child element is not included in the output document.

Sadly, there is one major problem with the previous example. All other elements are output using their text values, the element start and end tags and attributes have been lost from the document. The problem can be overcome using the `<xsl:copy>` element.

The `<xsl:copy>` element is used to copy elements from the XML source to the output document. The following rule is an identity transformation rule (the document is copied without any changes):

```
<xsl:template  match="*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
    <xsl:apply-templates
➥select="*|@*|comment()|processing-instruction()|text()"/>
  </xsl:copy>
</xsl:template>
```

The template matches all elements and uses the `<xsl:copy>` element to copy the matched element. The body of the `<xsl:copy>` element must apply the template rules to the body of the matched XML node; otherwise, no output will occur.

The full stylesheet for transforming the old style DD into the new style is shown in Listing 17.15. The output from applying this stylesheet is shown in Listing 17.16.

LISTING 17.15 Full Text of `session.xsl`

```
   1: <?xml version="1.0"?>
   2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   3: <xsl:template match="session">
   4:   <xsl:element name="{./session-type}">
   5:     <xsl:apply-templates/>
   6:   </xsl:element>
   7: </xsl:template>
   8: <xsl:template match="session/session-type"/>
   9: <xsl:template  match="*|@*|comment()|processing-instruction()|text()">
  10:   <xsl:copy>
  11:     <xsl:apply-templates
➥select="*|@*|comment()|processing-instruction()|text()"/>
  12:   </xsl:copy>
  13: </xsl:template>
  14: </xsl:stylesheet>
```

LISTING 17.16 Applying `session.xsl` to `dd.xml`

```
   1: >java org.apache.xalan.xslt.Process -in XML/dd.xml -xsl XSL/session.xsl
   2: <?xml version="1.0" encoding="UTF-8"?>
   3: <!-- DOCTYPE ejb-jar PUBLIC
➥'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
➥'http://java.sun.com/dtd/ejb-jar_2_0.dtd' -->
   4: <ejb-jar>
   5:   <display-name>Agency</display-name>
   6:   <enterprise-beans>
   7:     <Stateless>
   8:       <display-name>AgencyBean</display-name>
   9:       <ejb-name>AgencyBean</ejb-name>
  10:       <home>agency.AgencyHome</home>
  11:       <remote>agency.Agency</remote>
  12:       <ejb-class>agency.AgencyBean</ejb-class>
  13:
  14:       <transaction-type>Bean</transaction-type>
  15:     </Stateless>
  16:     <Stateful>
  17:       <display-name>AdvertiseBean</display-name>
  18:       <ejb-name>AdvertiseBean</ejb-name>
  19:       <home>agency.AdvertiseHome</home>
  20:       <remote>agency.Advertise</remote>
  21:       <ejb-class>agency.AdvertiseBean</ejb-class>
  22:
  23:       <transaction-type>Bean</transaction-type>
  24:     </Stateful>
  25:   </enterprise-beans>
  26: </ejb-jar>
```

Attributes and Attribute Sets

In the example `jobSummary` document in Listing 17.3 and the stylesheet `table.xsl` in Listing 17.10, you might like to be able to use a hyperlink to link the customer name to a URL of the form

```
/agency/advertise?customer=<customer_name>
```

Using the sample data from Listing 17.3, the output heading for a job for `winston` would look as follows:

```
<H2>Job ref: <A HREF="/agency/advertise?winston">winston</A>/Cigar Trimmer</H2>
```

Here, the value of the new attribute is derived from the XML source document. This can be achieved by using the `<xsl:attribute>` element, as shown in the following:

```
<xsl:template match="job">
  <H2>Job ref:
```

```
    <A>
      <xsl:attribute name="HREF">
        /agency/advertise?<xsl:value-of select="@customer"/>
      </xsl:attribute>
      <xsl:value-of select="@customer"/>
    </A>/<xsl:value-of select="@reference"/></H2>
  <TABLE border="1">
  <xsl:apply-templates/>
  </TABLE>
</xsl:template>
```

The `<xsl:attribute>` element defines an attribute for the enclosing element (in this case, `<A>`). More than one attribute can be defined, but all attributes must be defined before any other text or child nodes in the element.

Sometimes, the same attributes must be defined for many different tags. This is a common requirement when applying styles to HTML tables. Consider making every cell in the table contain text with a blue sans-serif font. You could define the style separately for every cell in the table, but this is hard to maintain should the style requirements change. Alternatively, you could use Cascading Style Sheets (CSS), but the most popular browsers do not support CSS in a consistent manner. An XSL solution is to use an attribute set.

The `<xsl:attribute-set>` element defines the attributes you can apply to multiple elements. An `<xsl:attribute-set>` defining a blue sans-serif font is as follows:

```
<xsl:attribute-set name="rowStyle">
  <xsl:attribute name="face">Arial,sans-serif</xsl:attribute>
  <xsl:attribute name="color">blue</xsl:attribute>
</xsl:attribute-set>
```

An attribute set is applied by using the `xsl:use-attribute-sets` attribute with an element. For example

```
<xsl:template match="skill|location">
  <TR><TD><FONT xsl:use-attribute-sets="rowStyle">
      <xsl:value-of select="name()"/>
  </FONT></TD><TD><FONT xsl:use-attribute-sets="rowStyle">
    <xsl:value-of select="."/>
  </FONT></TD></TR>
</xsl:template>
```

The transformed table definition for the job `skill` and `location` elements is as follows:

```
<TABLE border="1">
  <TR>
    <TD><FONT face="Arial,sans-serif" color="blue">location</FONT></TD>
    <TD><FONT face="Arial,sans-serif" color="blue">London</FONT></TD>
  </TR>
```

```
  <TR>
    <TD><FONT face="Arial,sans-serif" color="blue">skill</FONT></TD>
    <TD><FONT face="Arial,sans-serif" color="blue">Cigar maker</FONT></TD>
  </TR>
  <TR>
    <TD><FONT face="Arial,sans-serif" color="blue">skill</FONT></TD>
    <TD><FONT face="Arial,sans-serif" color="blue">Critic</FONT></TD>
  </TR>
</TABLE>
```

The complete stylesheet for transforming the `jobSummary` document into HTML is shown in Listing 17.17.

LISTING 17.17 Full Text of `tableStyle.xsl`

```
 1: <?xml version="1.0"?>
 2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 3: <xsl:template match="job">
 4:   <H2>Job ref:
 5:       <A>
 6:       <xsl:attribute name="HREF">
 7:         /agency/advertise?<xsl:value-of select="@customer"/>
 8:       </xsl:attribute>
 9:       <xsl:value-of select="@customer"/>
10:     </A>/<xsl:value-of select="@reference"/>
11:   </H2>
12:   <P><xsl:apply-templates select="description"/></P>
13:   <TABLE border="1">
14:   <xsl:apply-templates select="skill|location"/>
15:   </TABLE>
16: </xsl:template>
17: <xsl:template match="description">
18:   <I><xsl:value-of select="."/></I>
19: </xsl:template>
20: <xsl:attribute-set name="rowStyle">
21:   <xsl:attribute name="face">Arial,sans-serif</xsl:attribute>
22:   <xsl:attribute name="color">blue</xsl:attribute>
23: </xsl:attribute-set>
24: <xsl:template match="skill|location">
25:   <TR><TD><FONT xsl:use-attribute-sets="rowStyle">
26:       <xsl:value-of select="name()"/>
27:   </FONT></TD><TD><FONT xsl:use-attribute-sets="rowStyle">
28:     <xsl:value-of select="."/>
29:   </FONT></TD></TR>
30: </xsl:template>
31: </xsl:stylesheet>
```

Additional XSL Elements

XSL supports a number of elements that can provide program language like capabilities within a stylesheet. Using these elements requires a good knowledge of the XPath notation. Because you have only seen some of the XPath notation, you will only look very briefly at the elements that support programming capabilities.

Numbering Elements

A common requirement for many transformations is to insert numeric data into the output document, often to produce numbered lists.

The `<xsl:number>` element is used to insert numbers into the document. By default (that is, without attributes), this element inserts a count of the position of a context node in a list of elements of the same type.

To number the skills defined by a particular job, you could use the following rule:

```
<xsl:template match="skill">
  <TR><TD>
      Skill <xsl:number/>:
  </TD><TD>
    <xsl:value-of select="."/>
  </TD></TR>
</xsl:template>
```

Attributes can be supplied to the `<xsl:number>` element to determine what numeric value is inserted. The `level="any"` attribute is used to count all occurrences of the same type of node, regardless of where the node occurs in the document tree structure. The following example adds numbers to job definitions:

```
<xsl:template match="job">
  <H2><xsl:number level="any"/>.
      Job ref:
  …
</xsl:template>
```

Listing 17.18 shows the final stylesheet for transforming the `job.xml` document using numbers to count jobs and skills within a job. The resulting Web page is shown in Figure 17.3.

LISTING 17.18 Full Text of `tableCount.xsl`

```
   1: <?xml version="1.0"?>
   2: <xsl:stylesheet version="1.0"
➥xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   3: <xsl:template match="job">
   4:   <H2><xsl:number level="any"/>.
```

LISTING 17.18 Continued

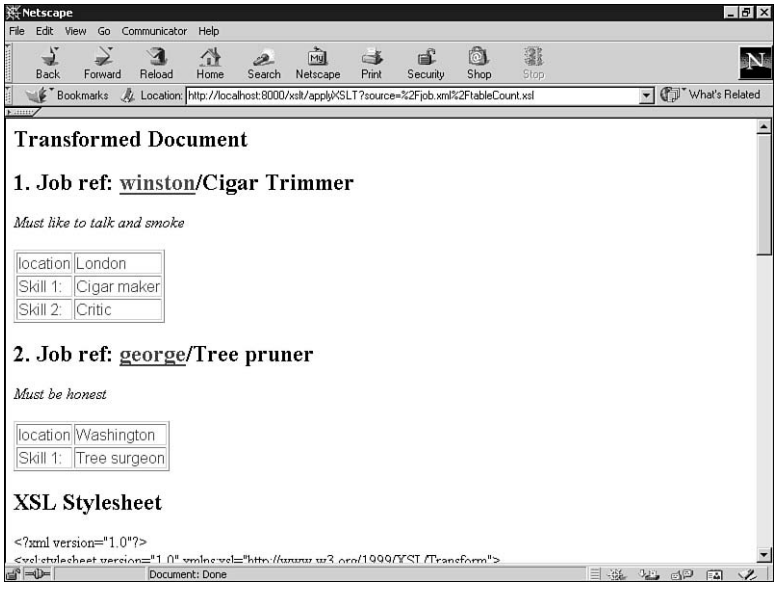
```
 5:       Job ref:
 6:       <A>
 7:       <xsl:attribute name="HREF">
 8:         /agency/advertise?<xsl:value-of select="@customer"/>
 9:       </xsl:attribute>
10:       <xsl:value-of select="@customer"/>
11:     </A>/<xsl:value-of select="@reference"/>
12:   </H2>
13:   <P><xsl:apply-templates select="description"/></P>
14:   <TABLE border="1">
15:   <xsl:apply-templates select="skill|location"/>
16:   </TABLE>
17: </xsl:template>
18: <xsl:template match="description">
19:   <I><xsl:value-of select="."/></I>
20: </xsl:template>
21: <xsl:attribute-set name="rowStyle">
22:   <xsl:attribute name="face">Arial,sans-serif</xsl:attribute>
23:   <xsl:attribute name="color">blue</xsl:attribute>
24: </xsl:attribute-set>
25: <xsl:template match="location">
26:   <TR><TD><FONT xsl:use-attribute-sets="rowStyle">
27:       <xsl:value-of select="name()"/>
28:   </FONT></TD><TD><FONT xsl:use-attribute-sets="rowStyle">
29:     <xsl:value-of select="."/>
30:   </FONT></TD></TR>
31: </xsl:template>
32: <xsl:template match="skill">
33:   <TR><TD><FONT xsl:use-attribute-sets="rowStyle">
34:       Skill <xsl:number/>:
35:   </FONT></TD><TD><FONT xsl:use-attribute-sets="rowStyle">
36:     <xsl:value-of select="."/>
37:   </FONT></TD></TR>
38: </xsl:template>
39: </xsl:stylesheet>
```

The `<xsl:number>` tag can also be used to

- Insert numeric data obtained from an element or attribute in the XML source document
- Count occurrences of a node
- Count nodes from a given start point
- Use letters or roman numerals instead of decimal integers or insert leading zeroes to make fixed-width numbers

FIGURE 17.3

Applying `tableCount.xsl` *to* `job.xml`.



Other Features

XSLT supports the optional inclusion of text in the output document using the following tags:

- `<xsl:if>` The body of this element is only included if the test defined as an attribute is `true`.
- `<xsl:choose>` Defines a list of choices, only one of which will be included.
- `<xsl:when>` Defines a choice for an `<xsl:choose>` element.
- `<xsl:otherwise>` Defines the default choice for an `<xsl:choose>` element if no `<xsl:when>` element is matched.

The `<xsl:if>` and `<xsl:when>` elements use a `test` attribute that evaluates an XPath expression and includes the element body if the test is `true`.

The following example tests if a Session bean from a DD is a stateless bean:

```
<xsl:if test="./session-type='Stateless'">
  …
</xsl:if>
```

The following `<xsl:choose>` example selects different transformations for stateful and stateless Session beans:

```
<xsl:choose>
  <xsl:when test="./session-type='Stateless'">
    …
```

```
  </xsl:when>
  <xsl:when test="./session-type='Stateful'">
    …
  </xsl:when>
  <xsl:otherwise>
    …
  </xsl:when>
</xsl:choose>
```

Other XSL elements include the following:

- `<xsl:sort>` Used to sort elements by alphabetic or numeric order
- `<xsl:include>` and `<xsl:import>` Import rules from other stylesheets
- `<xsl:variable>` Used to define variables that can be used in other XSL elements
- `<xsl:template>` Used to define templates that can be inserted in different parts of the transformed output (parameters can be used to customize each instance of a template)

As you can see, XSLT provides a powerful transformation language that really is too large and complex to cover in one lesson. Today's work on XSLT has been designed to give you an overview of what XSLT can do. For more information on XSLT, refer to the specifications on the W3C Web site or read the book *Sams Teach Yourself XML in 21 Days* from Sams Publishing.

XSLT Compilers

One drawback to XLST stylesheets is performance. An XLST processor, such as XALAN, must read in the stylesheet and build an internal structure representing the rules that must be applied. The processor must then read in the XML document and match each element to the rules defined in the stylesheet and generate the required output. All of this takes time.

One way to improve performance is to preprocess the stylesheet to create a custom program that will apply the one stylesheet to an XML document in an efficient manner.

Such a technology is called an XSLT compiler. This is a fast changing area of XSL technology, but the original XSLT compiler, called `xsltc`, developed by Sun Microsystems is now developed and maintained by the Apache project.

Apache provides the `xsltc` compiler used to compile an XSL stylesheet into a *translet* (a set of Java classes). An associated runtime processor is used to apply the compiled translet to an XML document and perform the XML document transformation.

An XSLT translet can be invoked from the command line or included in a developer's application.

The XLST compiler technology is very new and still maturing (and changing) as this book is being written in late 2001. For more information and to download the `xsltc` technology, take a look at the XALAN pages on the Apache Web site at

```
http://xml.apache.org/xalan/
```

Summary

Today you have looked at transforming XML documents into other data formats. The XSLT standard defines an XML stylesheet format that specifies how to transform an XML document into a new format. XSLT is commonly used to transform XML data into HTML for presentation by a Web browser.

An XSLT stylesheet defines a set of rules. Each rule

- Is matched against elements in an XML source document
- Defines transformations that are applied to the matched element to create the transformed data
- Can be applied to a selected element or multiple elements including a complete tree hierarchy of elements
- Uses the XPath notation to match XML elements (the same XPath notation is also used with XPointers in XML documents)

A new technology designed to address some of the performance problems of XSLT processors, such as Apache XALAN, is an XSLT compiler. An XSLT compiler has two components:

- A compiler than generates a translet (a set of Java classes) from an XSLT stylesheet
- A runtime processor that applies a translet to an XML document to perform the transformation

The XSL technology also identifies a portable device independent grammar (XSL-FO) for defining formatting requirements and the document data. XSL-FO is not being widely adopted by the industry at the present time.

You have now finished your excursion into XML. Tomorrow, you will return to Java programming and the J2EE platform to study Java and J2EE design patterns.

Q&A

Q What are the two components of XSL?

A XSLT defines a language that is used to write stylesheets that will transform an XML document into a different format.

Formatting objects, or XSL-FO, specifies a device-independent grammar for defining the format of a transformed XML page.

Q What are three techniques for applying an XSLT stylesheet for a Web client?

A

- Send the XML document and stylesheet to the client for processing.
- Convert the XML into an HTML file, store the file on the server, and send the file to the client to satisfy future HTTP requests.
- Transform the XML data on the server for every HTPP request for the data.

Q Which XSL element is used to insert the body of a matched XML tag into the output document expanding any nested elements?

A `<xsl:apply-templates/>`

Q What do the `.`, `..`, `//`, and `@name` XPath shortcuts expand to?

A `.` expands to `self::node()`
`..` expands to `parent::node()`
`//` expands to descendent-or-`self::node()`
`@name` expands to `attribute::name`

Q What are the XSL elements for inserting a comment, inserting an element, inserting an attribute, and copying elements?

A

- `<xsl:comment>`
- `<xsl:element>`
- `<xsl:attribute>`
- `<xsl:copy>`

Exercises

On Day 14, "JSP Tag Libraries," you developed a custom tag library for use with your Web application. Today, you will write an XSLT stylesheet that will transform a Tag Library Descriptor (TLD) document into an HTML page for easy viewing.

Listing 17.19 shows the `agency.tld` file from Day 14 that you will transform into HTML for display by a Web browser.

LISTING 17.19 Full Text of `agency.tld`

```
 1: <?xml version="1.0" encoding="ISO-8859-1" ?>
 2: <!--DOCTYPE taglib PUBLIC
 3:         "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 4:         "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd"-->
 5: <taglib>
 6:   <tlib-version>1.0</tlib-version>
 7:   <jsp-version>1.2</jsp-version>
 8:   <short-name>agency</short-name>
 9:   <tag>
10:     <name>getJob</name>
11:     <tag-class>web.GetJobTag</tag-class>
12:     <body-content>empty</body-content>
13:     <variable>
14:       <name-given>job</name-given>
15:       <variable-class>agency.AdvertiseJob</variable-class>
16:       <declare>true</declare>
17:       <scope>AT_BEGIN</scope>
18:     </variable>
19:     <attribute>
20:       <name>ref</name>
21:       <required>true</required>
22:       <rtexprvalue>true</rtexprvalue>
23:     </attribute>
24:     <attribute>
25:       <name>customer</name>
26:       <required>true</required>
27:       <rtexprvalue>true</rtexprvalue>
28:     </attribute>
29:   </tag>
30:   <tag>
31:     <name>getCust</name>
32:     <tag-class>web.GetCustTag</tag-class>
33:     <body-content>empty</body-content>
34:     <variable>
35:       <name-given>cust</name-given>
36:       <variable-class>agency.Advertise</variable-class>
37:       <declare>true</declare>
38:       <scope>AT_BEGIN</scope>
39:     </variable>
40:     <attribute>
41:       <name>login</name>
42:       <required>true</required>
43:       <rtexprvalue>true</rtexprvalue>
44:     </attribute>
```

LISTING 17.19 Continued

```
45:   </tag>
46:   <tag>
47:     <name>forEachJob</name>
48:     <tag-class>web.ForEachJobTag</tag-class>
49:     <body-content>JSP</body-content>
50:     <variable>
51:       <name-given>job</name-given>
52:       <variable-class>agency.AdvertiseJob</variable-class>
53:       <declare>true</declare>
54:       <scope>AT_BEGIN</scope>
55:     </variable>
56:     <attribute>
57:       <name>customer</name>
58:       <required>true</required>
59:       <rtexprvalue>true</rtexprvalue>
60:       <type>agency.Advertise</type>
61:     </attribute>
62:   </tag>
63:   <tag>
64:     <name>forEach</name>
65:     <tag-class>web.ForEachTag</tag-class>
66:     <body-content>JSP</body-content>
67:     <attribute>
68:       <name>collection</name>
69:       <required>true</required>
70:       <rtexprvalue>true</rtexprvalue>
71:       <type>java.util.Collection</type>
72:     </attribute>
73:   </tag>
74:   <tag>
75:     <name>option</name>
76:     <tag-class>web.OptionTag</tag-class>
77:     <tei-class>web.OptionTagTEI</tei-class>
78:     <body-content>empty</body-content>
79:     <attribute>
80:       <name>selected</name>
81:       <required>false</required>
82:       <rtexprvalue>true</rtexprvalue>
83:       <type>java.util.String[]</type>
84:     </attribute>
85:     <attribute>
86:       <name>default</name>
87:       <required>false</required>
88:       <rtexprvalue>true</rtexprvalue>
89:       <type>java.util.String</type>
90:     </attribute>
91:   </tag>
92: </taglib>
```

To display this TLD document, you will need to write a stylesheet that defines rules for the following transformations:

- Put the tag library name in a `<H1>` element.
- Define a `<TABLE border="1">` to contain all the tags.
- Highlight each tag name using a `<STRONG>` element.
- Put each tag in a row in the table and put the name, body content, all the attributes and all the variables each in their own cell.
- Put the attribute lists in their own table.
- Put the variable lists in their own table.

Listing 17.20 shows a suitable HTML page containing one row that meets the previously listed requirements.

LISTING 17.20 Transformed HTML Output

```
 1: <HTML>
 2: <HEAD>
 3: <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
 4: <TITLE>agency TLD</TITLE>
 5: </HEAD>
 6: <BODY>
 7: <H1>Tag Library Name: agency</H1>
 8: <TABLE border="1">
 9:   <TR align="left" valign="top">
10:     <TH colspan="2">Tag</TH><TH>Attributes</TH><TH>Variables</TH>
11:   </TR>
12:   <TR align="left" valign="top">
13:     <TH>Name</TH><TH>Body</TH><TH>Name Required</TH><TH>Name Class</TH>
14:   </TR>
15:   <TR align="left" valign="top">
16:     <TD><STRONG>getJob</STRONG></TD><TD>empty</TD><TD>
17:       <TABLE>
18:         <TR align="left" valign="top">
19:           <TD>ref</TD><TD>true</TD>
20:         </TR>
21:         <TR align="left" valign="top">
22:           <TD>customer</TD><TD>true</TD>
23:         </TR>
24:       </TABLE>
25:       </TD><TD>
26:         <TABLE>
27:           <TR align="left" valign="top">
28:             <TD>job</TD><TD>agency.AdvertiseJob</TD>
29:           </TR>
30:         </TABLE>
```

LISTING 17.20 Continued

```
31:         </TD>
32:       </TR>
33:     </TR>
34: </TABLE>
35: </BODY>
36: </HTML>
```

Finally, Figure 17.4 shows a screen snapshot of how your transformed HTML page should appear in a Web browser.

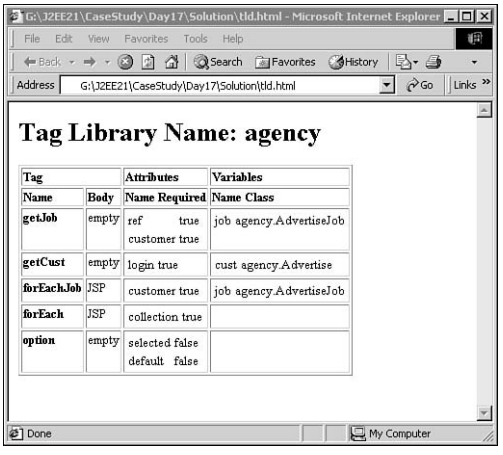


FIGURE 17.4
Applying

WEEK 3

DAY 18

Patterns

It has often been said that software engineering is more of an art than a science. Small teams, or even individuals, have been able to create sophisticated and powerful solutions using fairly ad-hoc design and development processes. On the other hand, many large projects that have been run on traditional engineering lines have arrived over budget and late, if they arrive at all. One of the main differentiators between successful and unsuccessful projects is the experience of those creating the system architecture and their understanding of the capabilities of the platform on which they are delivering that system.

The patterns movement within software attempts to capture some of the experience of successful architects and designers so that it can be applied more widely. The exploration of platform-specific patterns can also help designers to apply appropriate patterns in the right place using the right technologies. As J2EE technology has matured, J2EE-specific patterns have been discovered that should make your life as a J2EE designer and developer easier.

Today, you will

- Explore how the use of patterns can improve the design of systems
- Examine some of the J2EE-specific patterns currently documented

- Discover some of the J2EE patterns that are applied in the case study
- Suggest how other J2EE patterns can be used to improve the design of the case study

Today's intention is to investigate the role of J2EE-based patterns and see how such patterns can be applied in the context of the case study.

J2EE Patterns

Being a software designer, developer, or architect is not an easy job. To be truly effective, you must combine an understanding of the features required—the so-called problem domain—with a knowledge of the technologies and products from which the solution will be built. It is difficult for most practitioners to keep abreast of changing technologies in parallel with actually delivering applications to aggressive timescales. What is needed is some help to understand the best way to apply technologies and solve common design problems, rather than having to learn from trial and error. Patterns, and specifically in the context of this book, J2EE patterns, can help you to do this.

What Are Patterns?

To understand the intent of patterns, it is helpful to understand a little bit of their background.

The original inspiration of the software patterns movement is the work of the architect Christopher Alexander. His considerations of how we design buildings led him to identify and document common features of successful buildings. These common design features were christened *patterns*, and the documentation of a related set of them is termed a *pattern language*. Alexander's work can be explored in his writings, such as his book, *The Timeless Way of Building*.

An architect who designs buildings is constrained only by the qualities of the materials with which the building can be constructed together with cost constraints and certain physical laws (such as gravity, for example!). This leads architects to experiment with different forms of building, based on new materials and fashions in design. Some of these experiments are successful and turn up whole new ways of living and working. Many more experiments create buildings that are uninhabitable and are torn down in a relatively short time. Most architects need to build useful buildings that form a pleasant environment for those who live and work in them. The intention of Alexander's patterns is to provide a set of proven building blocks on which such architects can base their designs.

Now, consider the preceding paragraph in software terms.

An architect who designs software systems is constrained only by the technologies with which the system can be constructed together with cost constraints and certain physical laws (such as available bandwidth!). This leads architects to experiment with different forms of system, based on new technologies and fashions in design. Some of these experiments are successful and turn up whole new styles of application. Many more experiments create applications that are unusable and are discarded in a relatively short time. Most architects need to build useful applications that create a pleasant experience for those who live and work with them. The intention of software patterns is to provide a set of proven building blocks on which such architects can base their designs.

As you can see from this comparison, the motivations and intentions of architectural and software patterns have much in common.

So, what is a pattern? A short definition of a pattern is "A solution to a problem in a context." Essentially, a pattern is a reusable idea on how to solve a particular problem found in the domain of architecture or design, whether that be physical buildings or software systems. The key aspect of patterns is that they are proven solutions—you discover patterns, you don't invent them!

The concrete form of a pattern is a document describing certain aspects, including the following:

- A statement of the problem that the pattern addresses. This can include a list of conflicting requirements and issues that need to be balanced—known as *forces*.
- Contexts in which the pattern is known to work.
- A description of the solution, possibly including the detailed workings of the solution as it is applied in one context. For example, a software pattern may include code samples.

A pattern description may also set out different strategies within the pattern. These describe different approaches to solving the problem, possibly using different technologies or techniques, while still applying the same underlying principle.

Although patterns have common central aspects, there is no fixed way of documenting a pattern. Certain groups of patterns and pattern languages will adopt their own style of documentation that suits their needs. The main thing is that it is understandable and accessible.

If you want to investigate the general concept of patterns, the philosophy behind them, and the work of Christopher Alexander, try the following locations:

- "Christopher Alexander: An Introduction for Object-Oriented Designers" hosted at `http://g.oswego.edu/dl/ca/ca/ca.html`

- "Some Notes on Christopher Alexander" hosted at `http://www.math.utsa.edu/sphere/salingar/Chris.text.html`

Why Use Patterns?

The intent of patterns, then, is to provide a certain level of what might be termed "distilled wisdom." If people are lucky enough to work alongside a skilled architect or designer when they are learning their trade, they will gain such "wisdom" by osmosis. Patterns help the propagation of such wisdom so that the written pattern becomes a way of gaining some of the wisdom, even if you do not work alongside such a luminary.

The patterns become a toolchest for the designer or architect from which they can select. These tools can help them create an efficient, robust, and flexible solution. As such, the designer or architect must familiarize themselves with the patterns to the extent that they become adept at identifying the type of problem each pattern addresses and the pattern's context. In doing so, they can ensure that they select the right pattern or combination of patterns for the task at hand.

It is important to note that patterns are a guideline for a solution—not a solution in and of themselves. Although some design tools now provide templates for the implementation of common patterns, you cannot just drag-and-drop patterns into an application to form a solution. As a chef will adapt a recipe to suit the ingredients available and the taste of the guests, so a designer may adjust the implementation of a pattern to suit the specific context.

Types of Patterns

Patterns occur in all aspects of life, from spiders' webs to the way a school works. Some patterns are small and specific while others are general and wide-ranging. The same is true of the patterns found in the realm of software. The following are some common types of patterns:

- Architectural patterns define the overall style of the system and frequently include physical partitioning and infrastructure considerations as well as the software itself.
- Design patterns work at the level of software artefacts, such as classes and components, and describe ways in which such artefacts can be built and combined to solve common problems.
- Idioms are ways of solving a particular problem in a given environment, such as a language or platform. Some idioms are the embodiment of a pattern for a particular environment. For example, the Java mechanisms around the `Cloneable` interface and the reflection API are idioms that embody more general creational patterns.

- Process patterns have been identified relating to the actual mechanics of designing and creating the software and systems.
- Analysis patterns list common problems found in a particular business domain and ways in which these problems can be modelled.

The concept of patterns applied in the field of software was popularised mainly by the book *Design Patterns—Elements of Reusable Object-Oriented Software* written by Gamma, Helm, Johnson, and Vlissides. This is commonly referred to as the ”Gang of Four” book, or GoF for short. As its name suggests, this book contains a set of design patterns that are independent of platform and language. Many software patterns, including the J2EE patterns examined later, reference GoF patterns to help the reader understand aspects of the pattern being described.

Because patterns are found at different levels, the design of a system can involve the application of patterns inside patterns. An architectural pattern may define tiers or services from which the system is built. The contents of these tiers and services may be specified in terms of groups of components that conform to design patterns. These components, in turn, may be implemented using language- or platform-level idioms.

Sets of interlocking patterns have been discovered that form a toolchest for a particular context or at a particular level. These pattern languages help to promote consistent and coherent use of patterns. One such context is the J2EE platform, and it is a pattern language for this environment that will be explored today.

J2EE Patterns

The term *J2EE patterns* is used to refer to a set of patterns that have been identified within J2EE-based solutions. These patterns describe how to apply J2EE technologies to address common problems found when creating modern, distributed systems. As such, they fit the criterion stated earlier that they are “a solution to a problem in a context.” In the case of J2EE patterns, that context is the J2EE platform and the typical application architectures used when building J2EE applications.

Some patterns come with the territory in that they are found within the J2EE platform itself. However, the term J2EE patterns tends to refer to patterns that can be applied when creating systems on top of the J2EE platform.

Some J2EE patterns are simply direct implementations of previously identified patterns using J2EE technologies. An example of this would be the use of publish/subscribe when applying JMS. Other patterns are specific adaptations of known patterns for J2EE-oriented issues. An example of this is the Session Facade pattern, discussed later in the section on “Session Facades and Entity EJBs,” that encourages correct use of Entity EJBs.

Given that all J2EE patterns share a common context (namely, J2EE), they form a pattern language within that context. Some J2EE patterns are built on concepts from J2EE patterns or include them as part of a suggested implementation.

The set of J2EE patterns laid out here (and in other places) does not form a closed set from which J2EE applications can be built. An architect or designer can apply generic patterns or architectural patterns as they see fit when creating a J2EE-based solution. The J2EE-specific patterns simply provide a known set of J2EE-oriented solutions.

Pattern Catalogs

Patterns tend to evolve. There is no central body for the approval of patterns. A pattern tends to become accepted over time (or not, as the case may be) by the general audience or designers and architects at which it is targeted. Given this, how do you find patterns that you can apply?

There are various places where you can find information on patterns:

- There are now many general patterns books and repositories. These include the GoF book, the Pattern-Oriented Software Architecture (POSA) book series, the Pattern Languages of Programming (PLoP) book series, and the Hillside online pattern resources hosted at `http://www.hillside.net/`.
- J2EE-specific pattern information is available through the Sun Java Center patterns documented in the *Core J2EE Patterns* book. The Sun J2EE Blueprints patterns are available at the J2EE Blueprints Web site (`http://java.sun.com/blueprints/enterprise/`), the proposed J2EE patterns are available at `TheServerSide.com (http://www.theserverside.com/patterns/`, and many patterns with J2EE-related content are available on IBM's DeveloperWorks Web site at `http://www.ibm.com/developerworks/patterns/`.
- Some products, such as Together/J, Rational Rose, and Aonix Component Factory, provide templates and documentation to help you build systems based on common patterns.

In all cases, the patterns are embodied in documentation and examples.

Applying J2EE-Specific Patterns

Now that you understand the rationale and intent of patterns, how can you apply them to your J2EE applications? To do this, you must understand the J2EE design context, and you must also become familiar with the patterns themselves.

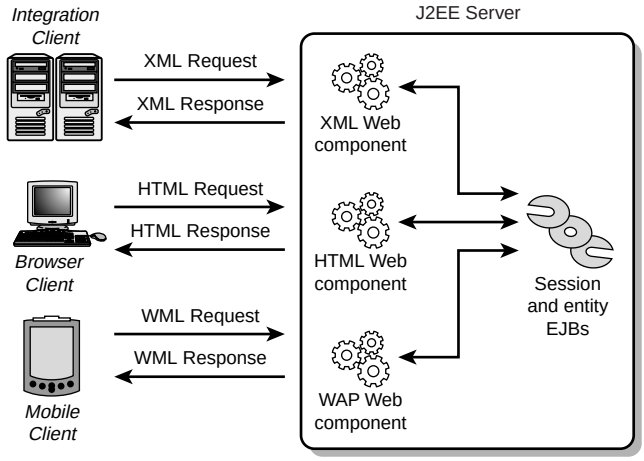
Applying Patterns in a Context

Most of the patterns defined for the J2EE platform range between the design and architecture levels. The most common J2EE architecture is largely assumed—a 3-tier business system—as the context for these patterns. You may be able to apply individual patterns in different styles of architecture, but as a pattern language, the current J2EE patterns are squarely targeted at a 3-tier, Web-based business system. The 3 tiers are usually classified as the presentation or Web tier, the middle or business tier, and the integration or data tier.

As well as consisting of multiple tiers, the target architecture is roughly based on the Model-View-Controller (MVC) principle. MVC is a form of pattern that can be applied at several levels. At the GUI level, it splits the responsibilities for interacting with the user between a data model, a presentation-oriented view, and a controller to govern the interaction. At the architectural level, this translates into entity EJBs (and various other components) providing a data model, servlets and JSPs providing the view, and Session EJBs providing the controller or business logic. Separating concerns in this way delivers a lot of flexibility. An example of this is that the model and controller (the data and the business logic) can be combined with a variety of views to expose the same functionality to different clients. This is shown in Figure 18.1.

FIGURE 18.1

Multiple views for different types of client within an MVC architecture.



Given this context, and given that an overall application style is in place, why do you need more patterns? Well, the purpose of applying patterns within an n-tier application generally relates to the systemic qualities of the application. *Systemic qualities*, sometimes called non-functional requirements, refer to qualities such as maintainability, extensibility, scalability, availability, and so on. By applying the patterns outlined today, you should be able to improve at least one of these systemic qualities in your application, if

not several. When the case study is examined later, the impact on systemic qualities of the patterns applied is also considered.

If you are designing an application from scratch, it will be second nature to think of applying patterns to the design process. However, many times you will be working within a pre-defined architecture or with an existing application. This does not mean that patterns no longer apply. It is quite possible to apply patterns to existing applications to improve them. It may be that the original design was not well thought out, or that it used the technologies in a naïve way (quite common when technologies are new). A pattern can be retro-fitted to part of the application to improve the systemic qualities of the application and to generally clean it up. This process is called *refactoring*, and is a key element in many software processes. As part of examining the case study, several potential refactorings will be considered.

The last thing that you need before examining the case study is knowledge of the patterns themselves.

Generic Patterns

Table 18.1 lists some common, generic patterns that are documented in the GoF book.

TABLE 18.1 Common GoF Patterns

Pattern Name	*Pattern Description*
Proxy	Provide a surrogate for another object or component to control access to it or enable access to it
Decorator	Add a variable level of functionality dynamically with the ability to plug in or remove components or filters as required (sometimes also known as wrapper or pipes and filters)
Singleton	Provide a single instance of a component and a global point of access to it
Iterator	Provide sequential access to a collection of objects in a way that is independent of the underlying representation
Observer	Define a relationship between components so that a change in the state of one of them causes a notification of this change to be delivered to the others
Façade	Provide a unified interface for a subsystem, thereby hiding underlying complexity
Command	Encapsulate a request with its data so that it can be presented and executed as a whole, without having to specify many different processing methods

J2EE Presentation-Tier Patterns

Table 18.2 lists patterns that have been identified around the presentation (or Web) tier of an n-tier J2EE application. The origin of each pattern is denoted using initials—SJC (Sun Java Center), BLU (Sun J2EE Blueprints), TSS (`TheServerSide.com`).

TABLE 18.2 Common J2EE Presentation-Tier Patterns

Pattern Name	*Pattern Description*
Front Controller (SJC)	A servlet (or JSP) intercepts the request from the user and routes or "adds value" to the request.
Intercepting Filter (SJC)	Provide a Decorator-style (GoF) filter chain as part of a Front Controller.
View Helper (SJC)	Use a JavaBean or custom JSP tag to encapsulate functionality and separate Java functionality out of a JSP.
Composite View (SJC)	Compose a JSP from several different sub-components to provide a typical, multi-panel Web page view.
Dispatcher View (SJC)	A Front Controller (SJC) intercepts and routes (or dispatches) a request to a JSP (or view). The view or its View Helpers (SJC) retrieve the content and/or data required to populate the view.
Service to Worker (SJC)	A Front Controller intercepts and routes (or dispatches) a request to a JSP (or view). The Front Controller (SJC) (or its helpers) retrieves the content and/or data required and passes this to the view as JavaBeans.
Service Locator (SJC)	A client-side shared helper object caches frequently used EJB home interfaces and dispenses EJB remote references on request.

J2EE Business-Tier Patterns

Table 18.3 lists patterns that have been identified around the business (or middle) tier of an n-tier J2EE application. The origin of each pattern is denoted using initials—SJC (Sun Java Center), BLU (Sun J2EE Blueprints), TSS (`TheServerSide.com`).

TABLE 18.3 Common J2EE Business-Tier Patterns

Pattern Name	*Pattern Description*
Session Facade (SJC)	A Session EJB provides a Facade (GoF) to shield entity EJBs from direct client access and to obscure the data schema from the client.

TABLE 18.3 Continued

Pattern Name	*Pattern Description*
Business Delegate (SJC)	A client-side object hides EJB-specific (or JMS-specific) interaction and exposes local business-oriented methods.
Value Object (SJC)	Provide a snapshot of underlying data to be used as a convenient data parcel between client and server to avoid chattiness (excessive network traffic between client and server).
Value Object Builder (SJC)	A constructor of Value Objects (SJC) from disparate server-side data sources. It presents one Value Object-based interface for a set of varied business data.
Composite Entity (SJC)	Create a coarse-grained business entity from a set of fine-grained data objects, such as entity EJBs or DAOs.
Value List Handler (SJC)	Have a session EJB act as a data cache and provide single/multiple data element Iterator (GoF) capability.
Page-by-page Iterator (BLU)	A variant of Value List Handler (SJC).
Fast Lane Reader (BLU)	Retrieve data for reading directly from the database for speed, write data back via entity EJBs for transactions and consistency.

J2EE Integration-Tier Patterns

Table 18.4 lists patterns that have been identified around the integration (or data) tier of an n-tier J2EE application. The origin of each pattern is denoted using initials—SJC (Sun Java Center), BLU (Sun J2EE Blueprints), TSS (`TheServerSide.com`).

TABLE 18.4 Common J2EE Integration-Tier Patterns

Pattern Name	*Pattern Description*
Data Access Object (SJC)	Encapsulate data access behind a common interface that can be implemented in different ways for different data sources. Typically used for data access in servlets and Session EJBs to encapsulate direct database access.
Service Activator (SJC)	Allow an EJB to be called on receipt of an asynchronous message.
EJB Observer (TSS)	An Observer-based (GoF) event pattern providing a strategy using EJBs.

Patterns Within J2EE

As noted previously, certain patterns occur within the J2EE environment itself, including the following:

- The Proxy pattern (GoF) is used widely in J2EE. Examples include RMI stubs as client-side proxies and EJB objects as server-side proxies.
- An EJB home interface acts as a Singleton (GoF) for the creation of EJB instances.
- The servlet filters provided as part of J2EE 1.3 are a form of the Intercepting Filter pattern (SJC).

There are many other patterns that are applied within the Java and J2EE environments.

Patterns in Context

You now know some of the common patterns that can be applied in J2EE applications. Using this knowledge, you can analyse the case study followed throughout the book to see how they can be applied.

Analysing the Case Study

Patterns are usually best understood in context. It helps to understand the intention of the design (and its associated code) and what it is trying to achieve so you can understand why a particular pattern helps in that situation. By examining the code for the case study, this section intends to

- Identify some of the places where J2EE patterns have been applied in the case study
- Examine some of those patterns in detail, including how the pattern looks in code
- Consider what other patterns could have been applied in certain places and the changes that would be required to use those patterns
- Understand why other J2EE patterns are not relevant to the design of the case study

The intention is to look at the central patterns that occur (or could occur) within the case study. This section will not contain an exhaustive list of all the patterns used.

One important objective is to point out at each stage why the patterns are useful. This principally takes the form of indicating which systemic qualities the pattern improves, such as maintainability, extensibility, or scalability, and also any systemic qualities that may suffer due to the pattern being applied.

When examining any design, it is important to understand the context in which design decisions were made. In this case, it is important to understand that the case study is, in places, intentionally simplistic. There are two reasons for this:

- The case study is essentially a learning tool. Use of production-level code can sometimes obscure (or unnecessarily complicate) the underlying principles or steps required. Hence, various simplifications are made in places that result in sub-optimal design and code.
- The case study follows the same disclosure sequence as the chapters of the book. Every effort is made to ensure that technologies are not used before they are introduced. Because of this, some parts of the design use alternative mechanisms that, again, can be sub-optimal.

The following sections examine different aspects of the case study and the J2EE patterns within it.

Session Facades and Entity EJBs

Probably the most obvious use of a J2EE pattern in the case study are the Session Facades that prevent the clients from accessing the entity EJBs directly. This applies both for the standalone application clients and the servlets/JSPs. All of the session beans in the Agency (`Advertise`, `AdvertiseJob`, `Agency`, and `Register`) act in the role of Session Facade because they all manipulate data in entity EJBs based on requests from the clients.

An example of this relationship is shown in Figure 18.2. This figure shows the class relationships between an `AdvertiseClient`, its associated `AdvertiseBean` (the Session Facade), and the `CustomerBean` that holds the back-end data. The implementation of the `CustomerBean` reflects the nature of this relationship because it only has a local home interface. This means that the `CustomerBean` is only intended for use by other EJBs such as the `AdvertiseBean`.

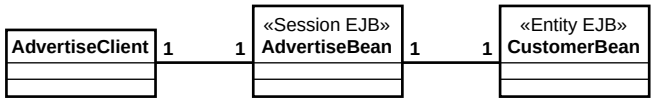


FIGURE 18.2

The `AdvertiseBean` acts as a Session Facade between the `AdvertiseClient` and the `CustomerBean`.

In terms of systemic qualities, the use of a Session Facade has the following impact:

- *Maintainability and flexibility*—By decoupling the client from the details of the underlying data model, there is a large reduction in dependencies. Fewer dependencies between layers make the system easier to change for maintenance reasons or in the face of changing requirements. The introduction of the Session Facade provides an ideal place for the location of business logic that is separated from both the client and the data management. Any changes to the business logic or the underlying data storage are then confined to the Session Facade and do not impact the client code.
- *Security*—There is now one point of access for a particular piece of business logic and its associated data. Appropriate controls can be placed on the Session Facade that are independent of the underlying data access control and properly reflect the logic being undertaken.
- *Performance*—In some ways, a Session Facade will reduce the level of performance because extra code and method calls are introduced. However, designing the Session Facade interface to offer a coarse-grained, business-oriented interface in place of the underlying entity bean's fine-grained, data-oriented interface can frequently offset any reduction in performance because it reduces the number of remote method calls required to perform a task. This type of interface simplification and business-orientation can be seen later when the use of a Value Object is discussed that reduces the number of remote method calls required to access and update a customer's details.

Before moving on, consider what you have just seen in terms of the general concept of patterns discussed at the start of this day's material. There are issues surrounding the use of Entity EJBs directly from remote clients. Most designers that have worked with EJBs for some time know of these issues. However, designers who are new to J2EE will not necessarily identify these issues. They may then produce poor quality systems until they learn about such issues by trial and error. By learning from such errors, the designer becomes better at his or her job (designing J2EE applications). However, this is of little solace to the owner of the poor application on which the designer learned his or her trade.

If designers know of the existence of J2EE-oriented patterns catalogs, they can study these as part of their J2EE education. They would then learn about the issues surrounding direct Entity EJB access from remote clients. If they read the description of the Session Facade pattern, they would know that they could introduce a Session EJB as described in this section. This has the effect of making the systems they produce more maintainable, flexible, and better performing. The designers of the Agency application

have used the Session Facade pattern where appropriate. Did they do this because they had learned from personal experience that direct Entity EJB access from distributed clients causes problems, or did they know this from studying J2EE patterns? To all intents and purposes it does not matter how they gained this J2EE design "wisdom," what matters is that the delivered system is of higher quality than it would be without the application of this pattern. By learning and applying the patterns described today, you should be able to improve the quality of the J2EE applications on which you work.

Data Exchange and Value Objects

Within the Agency application, the Session bean data access methods return collections of strings that identify jobs and other domain concepts. The client then uses one of these strings to create another Session bean to retrieve the details associated with that job. An alternative would be to use Value Objects to hold the information, such as job details, and to pass collections of these back and forth between the `Agency` Session bean and the clients.

The question of whether, how, and where to apply Value Objects revolves around the style of the application and the amount of data passed. When using an online catalog, such as those found in e-commerce applications, you will generally present the customer with a list of products. This list will be obtained as the result of some form of query (for example, all the books by Ian Fleming—or even all the products sold that relate to James Bond). The query results will be presented to the customer for him or her to make his or her choice. To make this choice, the customer needs more information about the product than just its name. Additional information could include its type (is it a video of *GoldenEye*, or a DVD, or a Playstation game, or a book?), the price, availability, and so on. This means that for every query result, you would want to pass back multiple pieces of information. In this case, encapsulating them in a Value Object would make a lot of sense. You could then return a collection of such Value Objects to the Web-tier client, which would then display them to the customer.

The style of the Agency application is somewhat different from this. Most of the information presented to the user is "top-level" information, such as a list of customers or locations. Information is retrieved and updated as required. Therefore, the application works more on the principle of "browse and drill down," meaning that the "next level" of information (such as the details for a particular customer) is only fetched when required after a selection has been made at the higher level. Passing back collections of Value Objects would be overkill in this situation because only one set of information (applicant or job) is required at any one time.

However, the Value Object pattern can be applied to several of the agency EJB interfaces. Take, as an example, the `Advertise` interface (shown in Listing 18.1) that is used to update or retrieve information on a particular job advertiser. Because the individual data items, such as the advertiser's name, must be retrieved individually through calls to the Session EJB, the application displays excessive chattiness. This means that there are lots of network connections, each one retrieving a small amount of data. This is in contrast to the `updateDetails` method that takes all of the advertiser's details in a single method call. You can apply the Value Object pattern to convert the repeated method calls into a single method call that returns a single JavaBean. This single JavaBean would contain all of the required information.

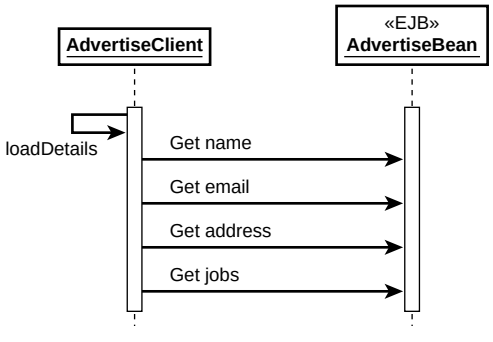
To show this pattern-based refactoring in practice, consider two forms of the `Advertise` interface. The original `Advertise` interface is shown in Listing 18.1 and has one method for each property defined on the advertiser. The interaction between the client and the Session EJB when loading the advertiser's details is shown in Figure 18.3.

LISTING 18.1 Original `Advertise` Interface

```
 1: public interface Advertise extends EJBObject
 2: {
 3:     void updateDetails (String name, String email, String[] Address)
 4:                                                 throws RemoteException;
 5:     String getLogin() throws RemoteException;
 6:     String getName() throws RemoteException;
 7:     String getEmail() throws RemoteException;
 8:     String[] getAddress() throws RemoteException;
 9:     String[] getJobs() throws RemoteException;
10:
11:     void createJob (String ref)
12:             throws RemoteException, DuplicateException, CreateException;
13:     void deleteJob (String ref) throws RemoteException, NotFoundException;
14: }
```


FIGURE 18.3

Interaction between the original `AdvertiseClient` and `AdvertiseBean`.



Listing 18.2 shows a refactored interface containing a single query method for the advertsiser's details that returns a Value Object of type `AdvertiseValueObject`. The interaction between the refactored client and Session EJB when loading details is shown in Figure 18.4. As you can see, there are now only two remote method calls made during the load rather than four, thus reducing the chattiness of this part of the application. This change will also approximately halve the time taken to load the details, because the time taken to make the remote calls will generally dwarf the time spent in local processing.

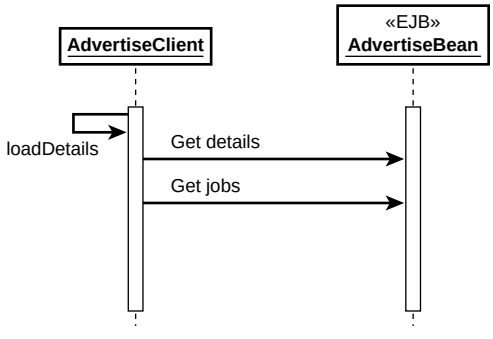
Note that you can now also use the Value Object as a parameter to `updateDetails` when updating the advertiser's details. Although this change to the update does not improve network performance, it does mean that any changes to the information held per-advertiser need only be made to the Value Object and the code that manipulates Value Objects rather than changing the `Advertise` interface.

LISTING 18.2 A Refactored `Advertise` Interface

```
 1: public interface Advertise extends EJBObject
 2: {
 3:     void updateDetails (AdvertiseValueObject details) throws
RemoteException;
 4:     AdvertiseValueObject getDetails() throws RemoteException;
 5:
 6:     String[] getJobs() throws RemoteException;
 7:
 8:     void createJob (String ref)
 9:             throws RemoteException, DuplicateException, CreateException;
10:     void deleteJob (String ref) throws RemoteException, NotFoundException;
11: }
```

FIGURE 18.4

Interaction between the refactored `AdvertiseClient` and `AdvertiseBean`.



Listing 18.3 shows a possible implementation for the `AdvertiseValueObject`. This is a very simple implementation and it would be quite possible to improve it, such as by providing multiple constructors or allowing smarter addition of address information. However, the code shown is the minimum you would need. Note that the Value Object conforms to the rules for a JavaBean in that it has a no-argument constructor, it uses getter/setter naming, and it is declared as implementing `Serializable`.

LISTING 18.3 `AdvertiseValueObject`

```
 1: public class AdvertiseValueObject implements java.io.Serializable
 2: {
 3:   private String _login;
 4:   private String _name;
 5:   private String _email;
 6:   private String[] _address;
 7:
 8:   public AdvertiseValueObject() {}
 9:
10:   public String getLogin() { return _login; }
11:   public void setLogin(String login) { _login = login; }
12:   public String getName() { return _name; }
13:   public void setName(String name) { _name = name; }
14:   public String getEmail() { return _email; }
15:   public void setEmail(String email) { _email = email; }
16:   public String[] getAddress() { return _address; }
17:   public void setAddress(String[] address) { _address = address; }
18: }
```


Using the `AdvertiseValueObject`, the code in the client (`AdvertiseClient.java`) that loads the details would change to that shown in Listing 18.4. The code has changed very little, but the bulk of the data retrieval methods (apart from `getDetails` and the `getJobs` method that is used in the `loadJobs` method) are now local rather than remote.

LISTING 18.4 Refactored `loadDetails` Method

```
 1: ...
 2: public class AdvertiseClient ...
 3: {
 4:   ...
 5:   private void loadDetails (Advertise advertise) throws RemoteException
 6:   {
 7:     AdvertiseValueObject details = advertise.getDetails();
 8:
 9:     name.setText(details.getName());
10:     email.setText(details.getEmail());
11:     String[] address = details.getAddress();
12:
```

LISTING 18.4 Continued

```
13:     address1.setText(address[0]);
14:     address2.setText(address[1]);
15:
16:     loadJobs(advertise);
17:   }
18: }
```

One thing to note is that the list of jobs associated with the advertiser is not included in the Value Object. The client manipulates the jobs separately from the rest of the advertiser details. Therefore, the updating and listing of jobs is kept separate from the manipulation of the advertiser's details.

In terms of systemic qualities, the use of a Value Object has the following impact:

- *Performance*—There is now one remote method call to retrieve applicant data rather than five. This will speed up the loading of the information by a sizeable factor (providing that there are no other performance roadblocks anywhere else).
- *Scalability*—Fewer calls across the network will reduce the amount of network bandwidth used and also the number of concurrent sockets required at each end of the connection. This means that the solution is more scalable (resources run out less quickly).
- *Maintainability and flexibility*—The reduction in dependencies and coupling brought about in the `Register` interface make subsequent changes to the applicant information easier to manage. Although changes to the contents of the Value Object will require changes in the client and server's internal data and the database schema, there is no need to change the remote interface.

There are also several variations on the Value Object pattern that you may find useful:

- *Partial Value Object*—If only part of the data held by the server is required by the client, a Value Object can be used that encapsulates precisely the data required. This will reduce the overall amount of data passed over the network.
- *XML Value Object*—You can extend the Value Object concept by passing an XML document instead of a Java object. This XML Value Object can be used to pass data between the presentation tier and the client tier. Alternatively, the EJB interface can pass the XML document as a `String` for easier interoperability with CORBA clients (see the CORBA discussion on Day 19, "Integrating with External Resources").

Data Access Without Entity EJBs

When you first started to look at the use of EJBs in the case study, on Day 4, "Introduction to EJBs," and Day 5, "Session EJBs," Entity EJBs were not yet used to

encapsulate the underlying data. Entity EJBs are a fundamental part of the J2EE architecture. They provide an extra level of flexibility by abstracting the underlying data source and, if using CMP, remove the need to write JDBC code. However, you may find that they do not bring any advantage for simpler, read-mostly applications. In this case, you may want to stick with direct database access from Session EJBs.

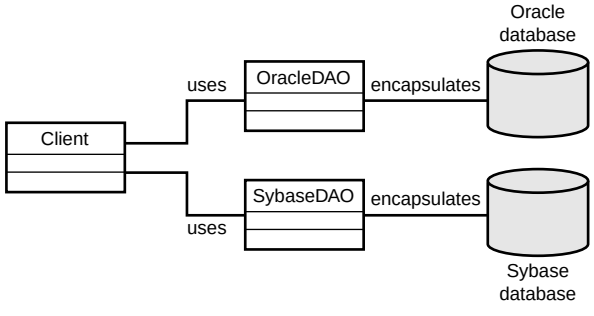
The Session EJBs from the Day 5 case study use direct database access for all queries and updates. Although direct database access is conceptually simpler than using an Entity EJB, it does mean that the data access code is intermingled with the business logic in the Session bean. This has a negative impact on

- *Maintainability*—Changes to either the data access code or the business logic will require that you change the Session bean code. An incorrect update can potentially destabilize all of the functionality of the Session bean.
- *Flexibility*—Should the underlying data source change, such intermingled code is difficult to change. This change could be anything from the use of a different database to a completely different data storage mechanism, such as LDAP, mainframe, or Entity EJBs.

To overcome these issues, you could apply the Data Access Object (DAO) pattern to these Session EJBs that separates the data access code from the business logic. Using a DAO to house the data access code means that the underlying data source becomes *pluggable*. The user of the DAO delegates all responsibility for the specifics of data access, such as mechanism, location, and error handling, to the DAO implementation. An example of this pluggability is shown in Figure 18.5.

FIGURE 18.5

A DAO acts as an adapter between the user and the specific data source.



To achieve this level of pluggability, the data and operations associated with the underlying data store must be abstracted. A J2EE DAO will use three mechanisms for this:

- The DAO defines a Java interface that will be implemented by all forms of the DAO. This interface will provide creation, retrieval, update, and deletion (CRUD) methods in much the same way that an EJB home interface does.

- One or more Value Objects represent the data to be used with the CRUD methods. Collections of Value Objects represent returned data. Because the Value Object is an abstraction of the underlying data, this helps to decouple the user of the data from the underlying data access mechanism, such as a JDBC `ResultSet`.
- Optionally, a factory can be used to determine the type of DAO used. This allows the user of the DAO to leave the implementation selection until runtime.

So, how could you apply a DAO in the pre-entity agency? Well, the first thing to do would be to decide on the granularity of the DAO or DAOs to be used. You could potentially create one large DAO to represent all of the underlying data in the system, but this would be somewhat unwieldy and difficult to maintain. Hence, the cleanest design would be to create one DAO per-table (per-type of data) and then see how that worked. Should this cause performance problems, the DAOs could potentially be merged or optimised later.

To illustrate the application of a DAO, consider the two main data types used by the `Advertise` Session EJB—job and customer. Focusing on the job information, the first thing to do would be to define the interface and Value Object to be used by a potential job-related DAO. Listing 18.5 shows a DAO interface, `JobDAO`, that could be used for this purpose.

LISTING 18.5 `JobDAO`, a DAO Interface for Use with Job Information

```
1: public interface JobDAO
2: {
3:   public Collection findByCustomer(String customer) throws Exception;
4:   public void deleteJob(String ref, String customer) throws Exception;
5:   public void createJob(String ref, String customer) throws Exception;
6: }
```

The methods to create and delete jobs are fairly self-explanatory. The `findByCustomer` method returns a polymorphic `Collection` that will contain Value Objects representing all the jobs found that are associated with the given customer. An example of such a Value Object is shown in Listing 18.6. This time, the Value Object has several read-only properties that are set when it is initialized.

LISTING 18.6 `JobValueObject`, a Value Object for Use with Job Information in a Job DAO

```
1: public class JobValueObject implements java.io.Serializable
2: {
3:   private String _ref;
4:   private String _customer;
```

LISTING 18.6 Continued

```
 5:   private String _description;
 6:   private String _location;
 7:
 8:   public JobValueObject(String ref, String customer,
➥String description, String location)
 9:   {
10:     _ref = ref;
11:     _customer = customer;
12:     setDescription(description);
13:     setLocation(location);
14:   }
15:
16:   public String getRef() { return _ref; }
17:   public String getCustomer() { return _customer; }
18:   public void setDescription(String description)
19:  {
20:     description = description;
21:  }
22:   public String getDescription() { return _description; }
23:   public void setLocation(String location) { _location = location; }
24:   public String getLocation() { return _location; }
25: }
```


The DAO interface and Value Object can then be used by an implementation of the DAO `DirectJobDAOImpl`. The implementation uses the JDBC code previously embedded in the Entity-less `AdvertiseBean`. Such a DAO is shown in Listing 18.7.

LISTING 18.7 `DirectJobDAOImpl`, a DAO Implementation that Uses JDBC Calls

```
 1: public class DirectJobDAOImpl extends DirectDAOImpl implements JobDAO
 2: {
 3:   public DirectJobDAOImpl(String jndiName)
 4:                                     throws SQLException, NamingException
 5:   {
 6:     super(jndiName);
 7:   }
 8:
 9:   public void createJob(String ref, String customer) throws Exception
10:   {
11:     PreparedStatement stmt = null;
12:     try
13:     {
14:       Connection connection = acquireConnection();
15:       stmt = connection.prepareStatement(
16:             "INSERT INTO Job (ref,customer) VALUES (?, ?)");
17:
18:       stmt.setString(1, ref);
```

LISTING 18.7 Continued

```
19:       stmt.setString(2, customer);
20:
21:       stmt.executeUpdate();
22:     }
23:     catch (SQLException e)
24:     {
25:         error("Error creating Job "+ customer +":"+ref, e);
26:     }
27:     finally
28:     {
29:       releasePreparedStatement(stmt);
30:       releaseConnection();
31:     }
32:   }
33:
34:   public void deleteJob (String ref, String customer) throws Exception
35:   {
36:     PreparedStatement stmt = null;
37:     Connection connection = null;
38:     try
39:     {
40:       connection = acquireConnection();
41:       connection.setAutoCommit(false);
42:       stmt = connection.prepareStatement(
43:             "DELETE FROM JobSkill WHERE job = ? AND customer = ?");
44:
45:       stmt.setString(1, ref);
46:       stmt.setString(2, customer);
47:       stmt.executeUpdate();
48:
49:       stmt = connection.prepareStatement(
50:             "DELETE FROM Job WHERE ref = ? AND customer = ?");
51:
52:       stmt.setString(1, ref);
53:       stmt.setString(2, customer);
54:       stmt.executeUpdate();
55:       connection.commit();
56:     }
57:     catch (SQLException e)
58:     {
59:       try
60:       {
61:         if (connection != null)
62:         {
63:           connection.rollback();
64:         }
65:       }
66:       catch (SQLException ex) {}
```

LISTING 18.7 Continued

```
      error("Error deleting job "+ref+" for "+customer, e);
    }
    finally
    {
      releasePreparedStatement(stmt);
      releaseConnection();
    }
  }

  public Collection findByCustomer(String customer) throws Exception
  {
    PreparedStatement stmt = null;
    ResultSet rs = null;
    Collection jobs = new TreeSet();
    try
    {
      Connection connection = acquireConnection();
      stmt = connection.prepareStatement(
            "SELECT ref FROM Job WHERE customer = ?");

      stmt.setString(1, customer);
      rs = stmt.executeQuery();

      jobs.clear();
      while (rs.next())
      {
        jobs.add(rs.getString(1));
      }
    }
    catch (SQLException e)
    {
      error("Error loading jobs for "+customer, e);
    }
    finally
    {
      releasePreparedStatement(stmt);
      releaseResultSet(rs);
      releaseConnection();
    }
    return jobs;
  }

  private void error (String msg, Exception ex) throws Exception
  {
     String s = "DirectJobDAOImpl: "+msg + "\n" + ex;
     System.out.println(s);
     throw new Exception(s +  ex);
  }
}
```

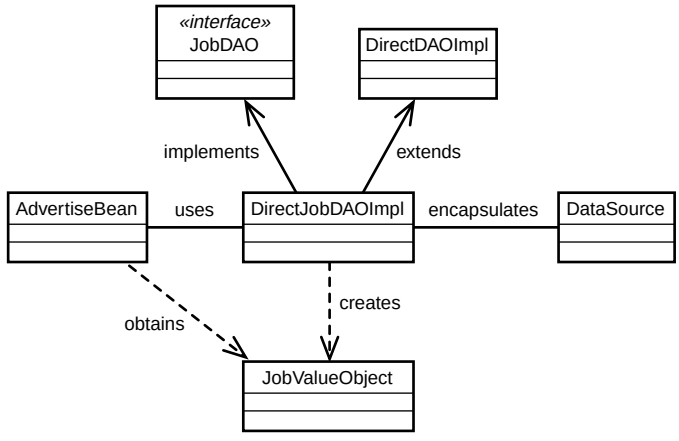
The DAO takes advantage of a superclass that manages the JDBC connection on its behalf. The subclass calls the methods `acquireConnection` and `releaseConnection` as required. All the rest of the JDBC manipulation is performed in the `DirectJobDAOImpl` class. The user of the class simply provides the JNDI string identifying the resource from which the data can be obtained. In this case, the resource will be the data source from which the `Agency` data can be obtained.

One thing to notice about the DAO implementation shown in Listing 18.7 is that it does not maintain a cache of jobs for the specific customer in the way that the Entity-less `AdvertiseBean` does. It would be quite possible to create a customer-specific form of the DAO that would cache this information to improve efficiency, if that was found to be beneficial.

Figure 18.6 shows the relationships between the different classes for this job DAO. The `AdvertiseBean` will instantiate a `DirectJobDAOImpl` and will call its methods to create, delete, and list jobs. This simplifies the code in the `AdvertiseBean` enormously and means that the code calls a single, meaningful method rather than containing many lines of JDBC calls and error handling.

FIGURE 18.6

Relationships between the classes that make up the job DAO.



The updated `createJob` method from the Entity-less `AdvertiseBean` is shown in Listing 18.8. You may notice that this code is now very similar to the code in the version of the `AdvertiseBean` that uses an Entity EJB. This is not surprising, because the DAO is essentially taking on the role of the Entity EJB in abstracting the data manipulation from the business logic.

LISTING 18.8 `createJob` Method from the Entity-less `AdvertiseBean` Updated to Work with a Job DAO

```
...
public class AdvertiseBean ...
{
  ...
  public void createJob (String ref)
                              throws DuplicateException, CreateException
  {
    try
    {
      jobDAO.createJob(ref, login);
    }
    catch(Exception ex)
    {
      error("Could not create job for " + ref, ex);
    }
  }
  ...
```

It is worth noting that in the Sun Java Center definition of Session Façade, the data objects protected from client access can be Entity EJBs, Session EJBs, or DAOs.

In terms of systemic qualities, the use of a Data Access Object has the following impact:

- *Maintainability and flexibility*—By separating out the data access from the business logic, it is possible to upgrade or replace the data access without affecting the business logic and vice versa. This makes the whole solution easier to maintain and evolve.
- *Performance*—As when adding any extra layer, there will be a certain reduction in performance due to extra object instantiations and method calls. However, these could potentially be offset by optimisations and caching in the DAO layer.

Messages and Asynchronous Activation

On Day 10, "Message-Driven Beans," message-driven beans were added to the `Agency` to match jobs with applicants and store the results of this matching process for later assessment. The reason for using asynchronous processing was due to the significant amount of time that this may take when the number of jobs or applicants has grown large.

The conversion of a synchronous EJB invocation into a message-driven form is defined by the Service Activator pattern. In EJB 2.0, this message receipt and processing can be easily implemented using message-driven beans. Prior to EJB 2.0, this pattern could be implemented by using a separate message server to process the messages and invoke the relevant EJB method.

In terms of systemic qualities, the use of a Service Activator has the following impact:

- *Performance*—Because the searching for matching jobs and applicants takes place asynchronously, the level of performance perceived by the client is better (in other words, the server returns immediately rather than "hanging" while the search is performed).
- *Scalability*—The client updates job or applicant information by calling a method on one of the relevant session EJBs, such as `AdvertiseJobBean`. Because each Session EJB will take up server resources (sockets, memory, and so on), the client should use the EJB as quickly as possible. After the client has finished using the EJB, its resources can be returned to the pool, ready to be used by another client. As a result, rapid processing leads to increased scalability because more clients can be serviced by the same number of resources in the same amount of elapsed time. The use of message-driven searching reduces the time that the client uses the Session EJB and so aids scalability.
- *Availability*—Should the server run out of message-driven beans in the pool (or should the server on which they run become unavailable), the messages can still be queued for delivery. If the service were accessed synchronously, a server outage would cause errors and stop the processing of client requests.

Composing an Entity

Although Entity EJBs are often used to represent rows in a database table, this mapping of table to Entity EJB is not necessarily optimal. If an Entity bean is directly based on an underlying database table, and then if the database table changes, so must the Entity EJB and all code that uses that Entity EJB. If there are many related tables, the application code may have to instantiate and use many Entity EJBs to access all of the data it needs. Each additional Entity EJB adds to the complexity of the application code. As more Entity EJBs are required, more RMI calls must be made to access their data, potentially increasing the traffic on the network and slowing down access to the data. Also, the application code may need to contact Entity beans from multiple data stores to retrieve all the data it requires.

To reduce the complexity for the application code and provide a more coherent view of the application's data, a single Entity can be used to represent data stored in multiple places (tables, databases, data stores). This single Entity bean can present a single, coarse-grained interface to the application code (usually a Session EJB).

In J2EE pattern terms, this is termed a *Composite Entity*. An example of this was seen in the `Agency` case study on Day 6, "Entity EJBs," when the `JobBean` was a BMP Entity EJB managing data from both the `Job` and `JobSkill` tables. Consequently, the `JobBean`

becomes a coarse-grained entity and any potential `JobSkill` Entity EJB is removed from the design. You performed the same design refactoring when you implemented the `ApplicantBean` to access both the `Applicant` and `ApplicantSkill` tables.

The `JobBean` and `ApplicantBean` BMP Entity EJBs both used JDBC directly to retrieve and update the required data. However, a good case can be made for combining Composite Entity with the DAO pattern so that the actual JDBC calls are handled in dependent DAO objects.

The Composite Entity pattern is applicable for both BMP and CMP Entity EJBs. However, for a CMP Entity, the specification of the dependent relationships can be done using many-to-many Entity relationships.

In terms of systemic qualities, the use of a Composite Entity has the following impact:

- *Maintainability and flexibility*—By reducing the number of entity EJBs, the application becomes easier to manage and maintain. Also, the use of one Entity EJB per-table effectively exposes the underlying database schema to the client. This makes it difficult to change at a later date. Providing access via a composite hides away the detail of the underlying data relationships.
- *Performance*—Any use of a composite interface as compared to multiple interfaces should reduce the chattiness of the application. However, some of this gain can be offset by a longer load time for the composite. The composite will potentially have to load more data than it needs to serve a simple request. However, there are several strategies, such as lazy loading, that can help to address this.

Composing a JSP

JSPs are easy to write and deploy. It is sometimes easy to forget that they are fully-featured and powerful Web components. Poor use of JSP functionality can have as much of an impact on the performance and scalability of your application as poor use of EJBs. J2EE Presentation Tier patterns, such as those listed in the "J2EE Presentation-Tier Patterns" section you saw earlier, can help you improve the quality of your Presentation Tier components and their interactions.

Consider some of the implications of using JSP functionality to deliver Web functionality. One common strategy when developing Web sites is to create one or more templates that define a standard layout for pages displayed to the user. As well as using common colors and styles, each page can display common information (the current location on the site) or functionality (a Home button) in a consistent place. Such templates help to improve the user interface of Web sites and make them easier to navigate. HTML frames have traditionally been used to divide the screen into separate areas to provide this templated look.

When developing JSP-based Web applications, the same principles of usability and navigability still apply. In the case of a JSP, the `@include` tag can be used to bring in standard functionality in specific parts of a page. The rest of the content in the page is determined by the JSP that is including the templated sections. This style of page composition using JSPs is captured in the J2EE Composite View pattern.

As an example of this, consider the `agency.jsp` from Day 13, "JavaServer Pages," the first few lines of which are shown in Listing 18.9. This JSP uses the `@include` tag to bring in the code from `header.jsf`, as shown in Listing 18.10. This header fragment brings in a set of standard page elements, both functional, such as the error page definition, and visible, such as the style sheet definition. The most visible part of this included fragment is the heading (in the `<H1></H1>` element) showing the agency name that can be seen at the top of Figure 18.7.

LISTING 18.9 Partial Listing of Day 13 `agency.jsp`

```
1: <html>
2: <head>
3: <title>Agency Portal</title>
4: <%@include file="header.jsf" %>
5: <%@page import="java.util.*" %>
6: <h2>Customers</h2>
7: <h3>Existing Customer</h3>
8: <form action=advertise>
9: ...
```

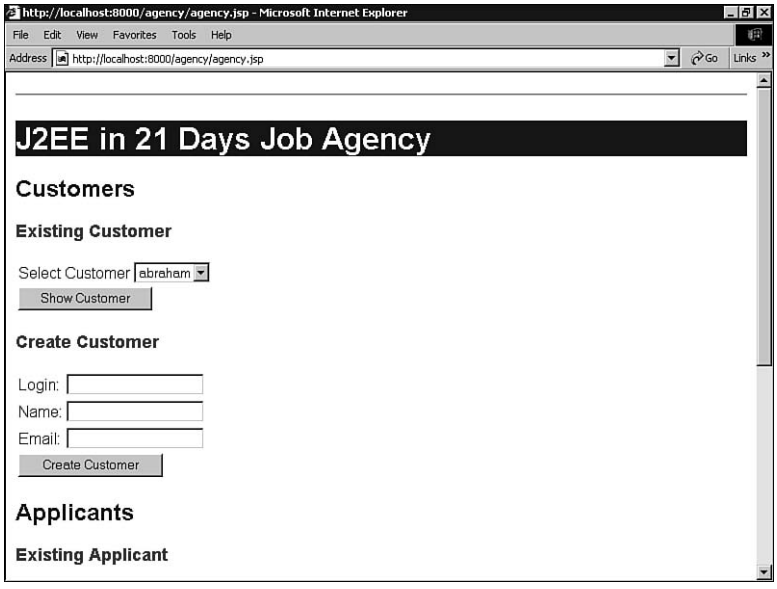
LISTING 18.10 `header.jsf`.

```
1: <%@page errorPage="errorPage.jsp" %>
2: <head>
2: <link rel=stylesheet type="text/css" href="agency.css">
3: </head>
4: <body>
5: <hr>
6: <jsp:useBean id="agency" class="web.AgencyBean" scope="request" />
7: <h1><jsp:getProperty name="agency" property="agencyName"/></h1>
8: <p>
```

The `header.jsf` fragment is included in all of the JSP pages that are provided for the Agency's Web interface. All of the other pages (apart from `agency.jsp`) also include the `footer.jsf` fragment at the bottom of the page. The `admin.jsp` page, shown partially in Listing 18.11, is an example of this. The `footer.jsf` fragment, shown in Listing 18.12, provides a button to take the user back to the main Agency JSP. The resulting admin page is shown in Figure 18.8.

FIGURE 18.7

The heading that shows the name of the Agency is part of a templated header included in `agency.jsp`.



LISTING 18.11 Partial Listing of Day 13 `admin.jsp`

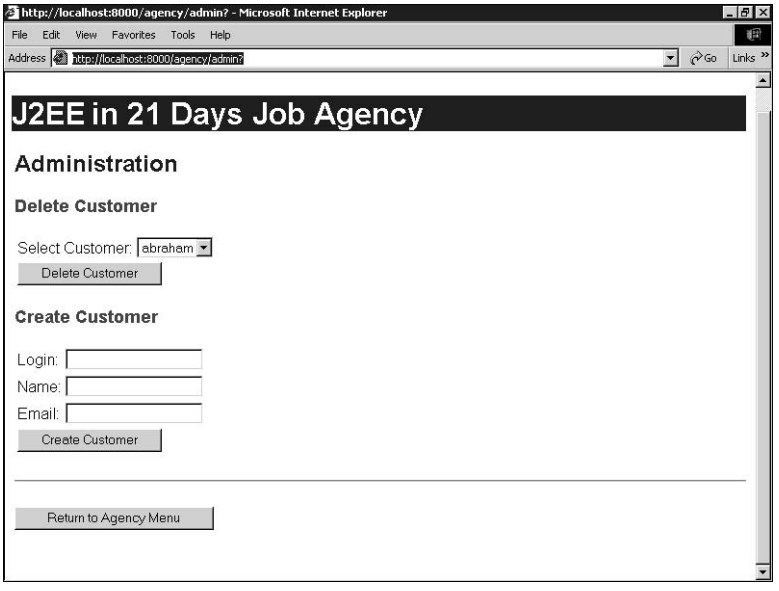
```
 1: <html>
 2: <head>
 3: <title>Agency Administration</title>
 4: <%@include file="header.jsf" %>
 5: <%@page import="java.util.*" %>
 6: <h2>Administration</h2>
 7: <h3>Delete Customer</h3>
 8: <form action=deleteCustomer>
 9: ...
10: </form>
11: <%@include file="footer.jsf" %>
12: </body>
13: </html>
```

LISTING 18.12 `footer.jsf`.

```
1: <p>
2: <hr>
3: <p>
4: <form>
5: <input type="button" value="Return to Agency Menu"
6:                                      onClick='location="agency"'>
7: </form>
8: <br>
```

FIGURE 18.8

admin.jsp includes both a header and a footer to provide consistent style and a standard button to return to the main screen.



The full Composite View pattern also includes a view manager component that allows parts of the page to be included dynamically. The view manager makes decisions about what should and should not be included in the page as the page is being generated. If the view manager takes the form of a JavaBean, conditional statements in the JSP can be based on method calls to the bean. An alternative strategy is to have the view manager implemented as a custom tag library. The tags themselves can then decide what should or should not be output for that page view.

In terms of systemic qualities, the use of a Composite View has the following impact:

- *Maintainability*—Because the template used for the pages is modular, particular parts of the template can be re-used throughout the application, giving a consistent look-and-feel, without having to maintain the same code in multiple pages.
- *Manageability*—Use of the pattern has benefits in that cleanly separated page fragments make it easier to make changes. If one part of the template is altered, this change is automatically propagated to all pages that include that part of the template. However, a plethora of fragments also presents a challenge to manage all of these pieces in a cohesive way.
- *Flexibility*—The use of a view manager to conditionally include fragments of content makes the application interface very flexible.

- *Performance*—The runtime generation of a page has a negative impact on performance, particularly when using a view manager. It may be best to have parts of the standard template pre-included when the JSPs are updated and refreshed in the Web application.

JSPs and Separation of Concerns

When designing an application, it is always important to correctly partition the components so that each component, or group of components, performs a specific task or tasks. This separation of concerns leads to fewer dependencies, cleaner code, and a more maintainable and flexible application. The separation of concerns becomes very important when those concerns relate to the skills of the people who must maintain and update parts of the application. In the case of JSPs, there are two distinct skillsets required—user interface design and writing J2EE-level Java code. Because it is rare to find these skills in the same person, these two aspects should be kept apart as much as possible.

Following this principle, the JSPs developed on Day 13 contain a certain amount of Java code to generate dynamic output, but not all of the code required is visible in the JSP pages. The JSPs use three JavaBeans, `AgencyBean`, `JobBean`, and `CustomerBean`, to perform the more involved processing required, such as the interaction with the business-tier EJBs. The J2EE pattern View Helper describes various ways in which encapsulated Java components can be used to hide a lot of the business processing required to create the desired output or view.

To see this in action, consider the `AgencyBean`, part of which is shown in Listing 18.13. As you can see, the constructor locates an `Agency` EJB (lines 31–37) and the rest of the code consists of methods that wrap calls to the EJB. If you look back to Listing 18.10, you can see that an `AgencyBean` declaration is included in `header.jsf` so that all pages will have an instance in scope. However, because the bean is scoped by request, the same bean can be shared if requests are forwarded to other JSPs or servlets within the application.

LISTING 18.13 Selected Highlights of `AgencyBean.java`

```
1: ...
2:
3: public class AgencyBean
4: {
5:   Agency agency;
6:
7:   public String getAgencyName() throws RemoteException
8:   {
9:     return agency.getAgencyName();
```

LISTING 18.13 Continued

```
 10:   }
 11:
 12:   public Collection findAllApplicants() throws RemoteException
 13:   {
 14:     return agency.findAllApplicants();
 15:   }
 16:
 17:   public void createApplicant(String login, String name, String email)
 18:                throws RemoteException, DuplicateException, CreateException
 19:   {
 20:     agency.createApplicant(login,name,email);
 21:   }
 22:
 23:   public void deleteApplicant (String login)
 24:                                   throws RemoteException, NotFoundException
 25:   {
 26:     agency.deleteApplicant(login);
 27:   }
 28:
 29:   ...
 30:
31:   public AgencyBean ()
 32:                 throws NamingException, RemoteException, CreateException
 33:   {
 34:     InitialContext ic = null;
 35:     ic = new InitialContext();
 36:     AgencyHome agencyHome =
 37:                         (AgencyHome)ic.lookup("java:comp/env/ejb/Agency");
 38:     agency = agencyHome.create();
 39:   }
 40: }
```

The use of the bean in the page makes it easy to build up JSP content based on the functionality exposed by the bean. The section of JSP code in Listing 18.14 is able to use the bean without the overhead of including the EJB-specific discovery code. This makes the page easier to maintain.

LISTING 18.14 Simple Use of the `AgencyBean` in the JSP

```
<form action=advertise>
<table>
<tr><td>Select Customer</td>
<td><select name="customer">
<% Iterator customers = agency.findAllCustomers().iterator(); %>
<% while (customers.hasNext()) {%>
  <option><%=customers.next()%>
<% } %>
```

LISTING 18.13 Continued

```
</select>
</td></tr>
<tr><td colspan=2><input type=submit value="Show Customer"></td></tr>
</table>
</form>
```

The relationships and interaction between the client, the JSP, and the bean are depicted in Figures 18.9 and 18.10. Everything to the right of the `AgencyBean` JavaBean is transparent to the `agency` JSP.

FIGURE 18.9

Class relationships for the `AgencyBean` View Helper.

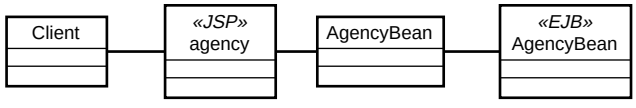
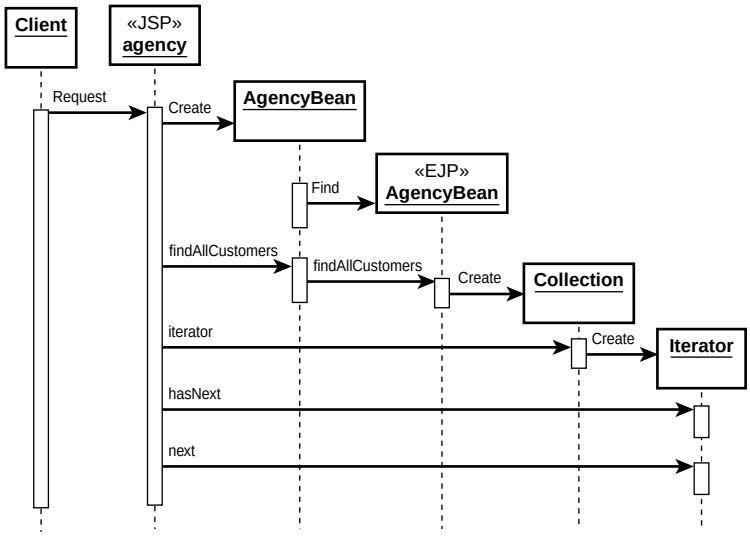


FIGURE 18.10

Example interaction diagram for the `AgencyBean` View Helper.



As well as using JavaBeans, there are other strategies for delegating code to components outside the JSP. The most powerful strategy is probably the one using a custom tag library. The code in Listing 18.15 shows how even more of the Java code is removed by using the custom `agency:forEach` and `agency:option` tags. This JSP code can now be easily manipulated and interpreted by a Web page designer and the tools used for Web page design.

LISTING 18.15 A Tag Library Used as a View Helper to Remove Code from the `Agency` JSP

```
 1: <form action=advertise>
 2: <table>
 3: <tr><td>Select Customer</td>
 4: <td>
 5:   <select name="customer">
 6:   <agency:forEach collection='<%=agency.findAllCustomers()%>'>
 7:     <agency:option/>
 8:   </agency:forEach>
 9:   </select>
10: </tr></tr>
11: <tr><td colspan=2><input type=submit value="Show Customer"></td></tr>
12: </table>
13: </form>
```

One other strategy for removing code from a JSP is to simply store the code in a JSP fragment and then include the fragment where appropriate. The `advertise.jsp` from Day 13 does this with `skills.jsf` and `location.jsf`. Although this can be effective in some cases, it lacks the encapsulation of the other mechanisms. It is important to note that the importing of these fragments relates more to the View Helper pattern than the Composite View pattern, as it may seem at first glance.

In terms of systemic qualities, the use of View Helpers has the following impact:

- *Maintainability*—If the code and the HTML of the JSP are separated, it makes it far easier to maintain the two parts. There is much less chance of accidental changes than when code and HTML are mixed.
- *Flexibility*—If the code behind the helpers changes, for example to use different EJBs or direct database access, there is no need for the JSP to be altered.
- *Performance*—As with any layering technique, there is the potential for some degradation in performance.

Client-Side Proxies and Delegates

The JavaBeans in the View Helper discussion also act as a shield between the JSP and the business tier functionality. For example, the JSP has no need to know that it is using an EJB when it instantiates an `AgencyBean` JavaBean. The JavaBean encapsulates all of the EJB handling (the custom tag library also does this in later versions). In this role of client-side proxy, the JavaBean provides an implementation of another J2EE pattern, the Business Delegate.

The role of a Business Delegate is to reduce the coupling between the client tier or Web tier code and the business tier implementation. As indicated, coupling is reduced by encapsulating the implementation of the business logic (as an EJB). This encapsulation

includes not only the lookup of resources but also error handling, retrys, and failover if appropriate. The Business Delegate can also perform caching and mapping of EJB exceptions to application exceptions.

When Business Delegates and Session Facades are used together, there is commonly a 1:1 mapping between them. This occurs in the Day 13 `Agency` code where the `AgencyBean`, `JobBean`, and `CustomerBean` JavaBeans each act as a Business Delegate for the `Agency`, `Job`, and `Customer` Session Facade EJBs, respectively. The primary reason that the Business Delegate does not contain business logic is that such business logic should be factored back into an associated Session Facade.

In terms of systemic qualities, the use of Business Delegates has the following impact:

- *Maintainability*—The Business Delegate encapsulates the business tier access code. Any changes required to this code can be made in one place.
- *Reliability and availability*—Retry and failover strategies can be implemented by a Business Delegate to improve the reliability and availability of the access to business tier services.
- *Performance*—As with any layering technique, there is the potential for some degradation in performance. However, a Business Delegate can also cache information if appropriate, having a positive impact on performance.

Locating Services

Because the case study is designed to be simple, it tends to obtain EJB home and remote references on a per-call basis. In a production system, this is one of the areas where caching would be of benefit. The Service Locator pattern describes a way to speed up the retrieval of EJB remote interfaces.

The Service Locator is a derivative of the GoF Singleton pattern. It acts as a central, common service on the presentation or Web tier from which EJB remote references can be obtained. The Service Locator will obtain and cache a reference to the home interfaces of the EJBs it serves. These cached home interfaces are then used to dispense EJB remote interfaces as required. The client code does not need to perform the home interface lookup every time and so becomes more efficient.

In terms of the case study, both the application clients and Web-tier components could use such a service to improve the way that they obtain EJB remote references.

In terms of systemic qualities, the use of Service Locator has the following impact:

- *Performance*—Caching the EJB home interfaces should improve the time taken to obtain an EJB remote reference markedly. Also, because there are fewer JNDI lookups, there will be less network traffic.

- *Reliability and availability*—Retry and failover strategies can be implemented by a Service Locator to improve the reliability and availability of the access to business tier services.
- *Maintainability*—Because all of the EJB lookup (or JMS queue/topic lookup) takes place in one component, there is one place for updates.

Any Other Business

Other J2EE patterns could be applied to the case study if the requirements were different. For example, the case study uses standard J2EE security in a fairly straightforward way. However, if there were more complex security requirements, a Front Controller servlet could be used in the Web tier to enforce such security in a uniform manner without having to place security code in each JSP and servlet. If other functionality, such as logging and audit, were required, the Intercepting Filter pattern could be applied to chain together all of the required common "transparent" functionality (security, logging, and so on) ahead of any access to JSPs and servlets.

Because the data displayed by the Web-based interface is fairly standard, there is no requirement for pre-processing of information in a Front Controller or for any form of routing to take place such that the client is shown different pages depending on the context information they submit (for example, a cookie or form field). If such pre-processing and conditional display were to form part of the case study, the patterns Service to Worker and Dispatcher View could be considered when deciding where to place the different responsibilities.

The case study is not an e-commerce system, as many J2EE systems are. Such systems tend to deal with catalogs of products and spend most of their time displaying static information and less time updating information (for example, a customer will browse many pages before placing an order). For e-commerce systems, other patterns in the J2EE catalog, such as Value List Handler and Fast Lane Reader, become relevant.

The different patterns are relevant in different contexts. Some patterns will be irrelevant in some cases, so don't assume that all patterns will be applicable in your system.

Refactoring the Case Study

As you have seen, there are various ways in which the case study can be refactored to improve its systemic qualities. The reason that such refactoring can be performed is down to the context in which the application was written, namely that parts of it were simplified for educational purposes.

As noted earlier, most development work is performed on existing code, so identifying a potential application of a pattern in such code and applying the appropriate refactoring is

a useful skill to build. The reason that you would want to apply the refactorings shown is to make the case study a more efficient, maintainable, and scalable application. But why are you doing that? Well, probably because you want to apply that application in the real world to serve real customers. In that case, the context for the application has changed from being an educational J2EE showcase to being a live production application. The change in context alters the requirements and the forces on which the application is based. Such changes will almost inevitably show up areas in which changes are required.

As any application is altered, whether that is the addition of new functionality or a change to the systemic quality requirements (for example, an increase in the order of magnitude of users), certain of the original design decisions may no longer make as much sense as they did in the original design context. As a result, refactoring of applications and the use of different patterns in different areas over time should become a way of life for software designers and developers. Without suitable maintenance, software will erode over time as the world changes around it.

Directions for J2EE Patterns

As you have seen, there are a number of patterns that have been identified as recurring in J2EE design and development. However, the list of J2EE patterns provided earlier is not a definitive and final one. More J2EE-specific patterns will undoubtedly emerge over time due to various factors:

- Because more applications are developed based on existing J2EE technologies, more patterns may be mined by identifying other common design elements within those applications.
- New patterns will evolve based on newly released technologies that become part of the J2EE platform. A prime example of this is in the area of Web Services. Web Services will play a major part in J2EE applications developed from J2EE 1.4 onward. The application of such new technology will give rise to new patterns of use. Because Web Services will form part of J2EE, these new patterns will find their place in the range of J2EE patterns available to the application designer.
- New applications of all these technologies will lead to new requirements. The solutions to these new requirements will give rise to new patterns of use and, hence, to new J2EE patterns.

Another aspect of the emergence of patterns in environments such as J2EE is that the common patterns become embodied in the environment itself. This was noted earlier, given that patterns such as Proxy are prevalent in the infrastructure and generated classes of a J2EE application. As J2EE evolves, more patterns will be captured in the underlying platform. An example of this is the introduction of servlet filters in J2EE 1.3, which embodies much of the Intercepting Filter pattern.

Summary

Good design relies on knowledge of the technologies in use and an appreciation of how they best fit together to solve particular problems. Technologies are comparatively straightforward to learn, and there are many sources from which you can obtain information and insight, such as books and training courses. Gaining an appreciation of how best to apply these technologies is more difficult. By studying design patterns, particularly those targeted at a specific platform such as J2EE, you can accelerate your understanding of the design issues in particular environments. Design patterns also provide you with a set of solutions to these issues that can be applied as part of your application design.

Q & A

Q There is much material available on application design and architecture. Why should I use patterns?

A Patterns are mined from concrete examples of design and architectural elements that have been used and proven in delivered systems. A pattern is only a pattern when it has been identified in multiple existing designs or architectures. As such, a pattern is more than just an opinion.

Q What effect do J2EE patterns have on the systemic qualities of a design, such as scalability and availability?

A Correctly applied, J2EE patterns will improve one or more of the systemic qualities of a design. As an example, consider the Value Object pattern. The primary purpose of using a Value Object between client and server is to reduce network chattiness, where a client makes repeated calls to a server to obtain a set of related data. By creating a single object to represent such data, only a single method call is required to retrieve the data. This improves the performance and scalability of the system.

Q Can I only apply J2EE patterns to new designs?

A No. J2EE patterns can be used to re-shape existing application code to improve its maintainability and other systemic qualities.

Q Why do J2EE patterns tend to abstract the access to underlying databases?

A An important part of good design is to reduce the dependencies between parts of a system. As an example, consider the use of a Session Facade to hide the underlying structure and relationships of the business data. The Session Facade interface should be business-oriented rather than data-oriented, because the Session Facade provides business services instead of raw data. Obscuring the structure of the data means that this data can be changed at a later date without requiring changes to the client.

Exercises

There are several refactorings that could be performed on the case study. One of the most effective is to use Value Objects between the clients and their associated Session beans. This exercise asks to you perform such a refactoring.

1. Create a Value Object called `ApplicantValueObject`. This should contain all of the information about an applicant, such as his or her name, e-mail, and so on. The class you create should be a JavaBean, and you must be able to use it as a parameter or return type in an RMI method call.
2. Alter the `Register` interface so that it passes `ApplicantValueObjects` between client and server. The interface should only have two methods.
3. Alter the `RegisterBean` EJB so that it implements the updated `Register` interface and uses `ApplicantValueObjects`.
4. Alter the `RegisterClient` application so that it uses the updated `Register` interface and `ApplicantValueObjects`.
5. Build the updated application and re-deploy it. Make sure that everything still works correctly.
6. An example solution is available under the `agency` directory in the Day 18 exercise code on the CD-ROM. Examine the sample solution in contrast to your own. How might you improve this solution? Areas to consider include the EJB home interfaces, the Value Object constructor, the way that the Value Object stores skills, and whether the `Applicant` EJB should create the Value Object.

WEEK 3

DAY 19

Integrating with External Resources

Yesterday, you learned about patterns and how they describe typical application development problems that you might face and how these patterns provide solutions to these problems. One problem that many application developers share is how to integrate a J2EE application with existing non-Java code, applications, or systems. Today's lesson introduces you to four possible solutions to this problem.

The first of these solutions is the J2EE Connector architecture, which allows you to connect to Enterprise Information Systems (EIS), such as Enterprise Resource Planning systems (ERP). Primarily, today's lesson focuses on this architecture, but it also shows you approaches to writing Java code that consumes non-Java code libraries (such as legacy C libraries) and consume remote objects, which are also not written in Java.

Today's lesson covers the following topics:

- Reviewing external resources and legacy systems
- Introducing the Connector architecture

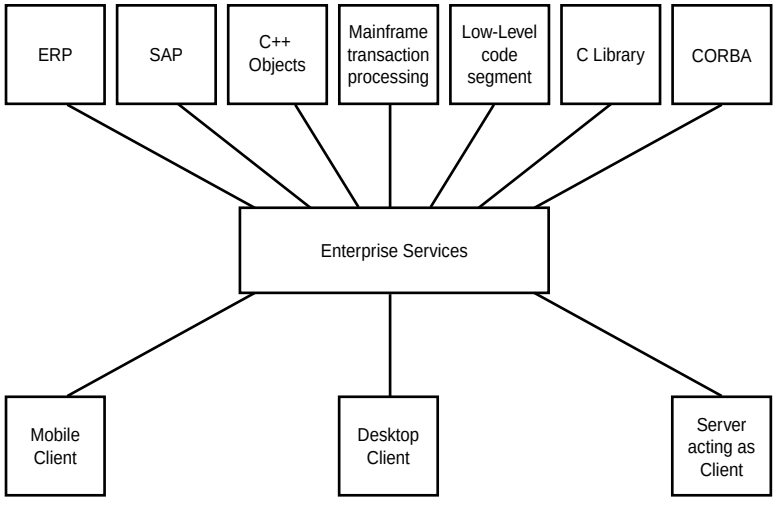
- Connecting to legacy systems by using the Common Client Interface
- Introducing Java IDL and CORBA
- Working with RMI over IIOP
- Working with the Java Native Interface
- Reviewing integration technologies

Reviewing External Resources and Legacy Systems

In an ideal world, your entire application might only contain Java components and applications, but it is not an ideal world. Very few architects have the opportunity to design entire Java systems from the ground-up, and very few enterprise developers have the opportunity to solely work on these types of systems. In reality, systems are heterogeneous in nature; that is, they are made up of many different parts. Figure 19.1 shows a complex heterogeneous environment that shows an array of clients attempting to access the services provided by a vast array of non-Java applications and systems.

FIGURE 19.1

A complex heterogeneous environment.



As you can see, the system consists of Enterprise Information Systems, such as ERP, SAP, and mainframe transaction processing systems. In addition, the enterprise services must call on the services provided by legacy code libraries, remote objects written in non-Java programming languages, and code written in lower-level programming languages.

The needs of business and the pressures of the application development cycle means that it is simply not feasible to rewrite all these elements so that they consist of pure Java. The only feasible option is that enterprise services provide mechanisms that integrate with these legacy and non-Java elements. Luckily, there are a number of Java technologies that allow you to integrate J2EE applications with these elements. It is these technologies that you will explore in today's lesson.

Introducing Connector Architecture

The Java solution to application integration in a heterogeneous environment is the J2EE Connector architecture. The architecture provides a standard way to connect to legacy and non-Java systems. As you will learn, the architecture allows EIS providers to create a single resource that allows any J2EE application server to access the EIS. In addition, the architecture defines a standard API that allows you to program against any supported EIS by using a standard set of API calls.

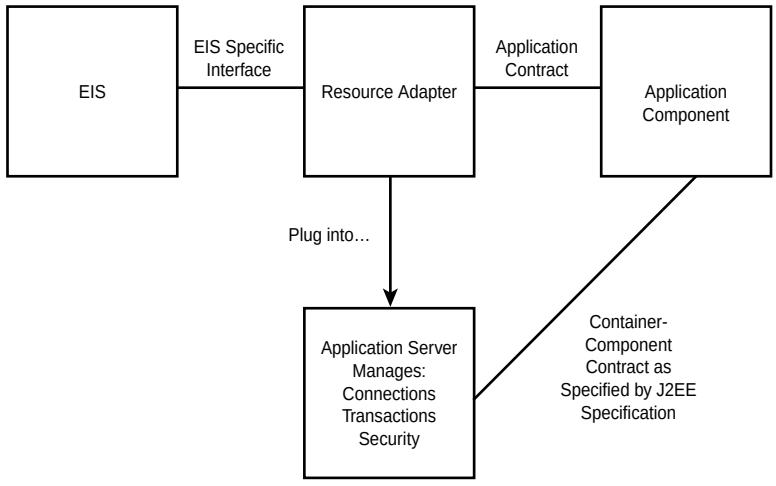
Today's lesson discusses and uses version 1.0 of the Connector architecture, but there is already a JSR proposing version 2.0 of the architecture. This version may include support for a number of items, including Common Client Interface metadata and XML integration. You can learn more about version 2.0 of the architecture by reading JSR112.

Overview of the Architecture

The J2EE Connector architecture defines two categories of contract—system and application. The system contract defines the relationship between a J2EE application server, or a container, and an EIS. The system-level contract requires the application server and the EIS to collaborate to hide system-level functionality from developers and components. The EIS provider implements its part of the contract through a resource adapter, which plugs into the application server to form a bridge between components and EISs. Figure 19.2 shows the resource adapter within the J2EE connector architecture.

Figure 19.2 also shows how application components, such as an EJB or JSP, communicate with an EIS through the resource adapter—not through direct communication. The application contract governs this communication through a client API—you will learn more about this contract later in the "Using the Common Client Interface" section of today's lesson.

FIGURE 19.2
The J2EE Connector architecture.



The `javax.resource.spi` package provides the classes, and interfaces that you require to write a resource adapter for an EIS. The process of writing a resource adapter is outside the scope of today's lesson. If you want to learn more about writing resource adapters, please refer to the J2EE API documentation and the Connector architecture specification.

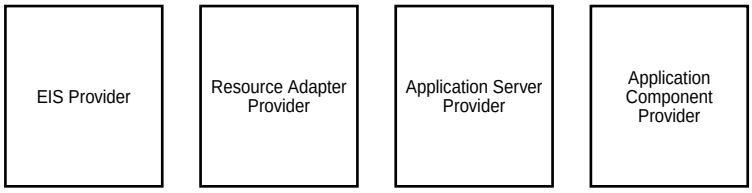
To draw a comparison with other elements of J2EE, you can consider a resource adapter to be similar to a JDBC driver. In both cases, a service provider or a third-party provides a bridge between a non-Java application and Java components. In the case of JDBC, the bridge is a JDBC driver and, in the case of the Connector architecture, the bridge is a system-level software driver known as a resource adapter. In both instances, components communicate to the underlying service via the bridge.

After an EIS provider has written a resource adapter, any J2EE-compliant application server can use that adapter. This means that the EIS provider only has to write one resource adapter to support many application servers. An application server can host several resource adapters—one for each EIS—and, thus, support multiple EISs. To offer you a deeper insight into the relationship between EIS, application server, and application component, the next section of today's lesson looks at the roles and responsibilities the Connector architecture specification defines. In addition, the next section takes a closer look at the service and application contracts.

Roles and Responsibilities

Figure 19.3 shows a simplified representation of the roles, which relate to the process Figure 19.2 outlines.

FIGURE 19.3
J2EE Connector architecture roles (simplified).



Typically, but not exclusively, the EIS provider and the resource adapter provider are the same person or organization. The resource adapter plugs into the application server, so that the server can provide application components with connectivity to the EIS. The application server manages security, resources, and transactions on behalf of the EIS, so that these system-level functions are transparent to the application component.

Previously, in Figure 19.2, you saw how a system contract governs the relationships between an EIS and an application server. The system contract comprises of three separate contracts:

- Connection contract
- Transaction Management contract
- Security contract

The remainder of this section of today's lesson examines each of these contracts, so that you can gain an understanding of the services available to you as an application developer. If you want to explore these areas in more depth, please refer to the J2EE Connector architecture specification, the current version of which is available at `http://java.sun.com/j2ee/connector/index.html`.

The Connection Contract

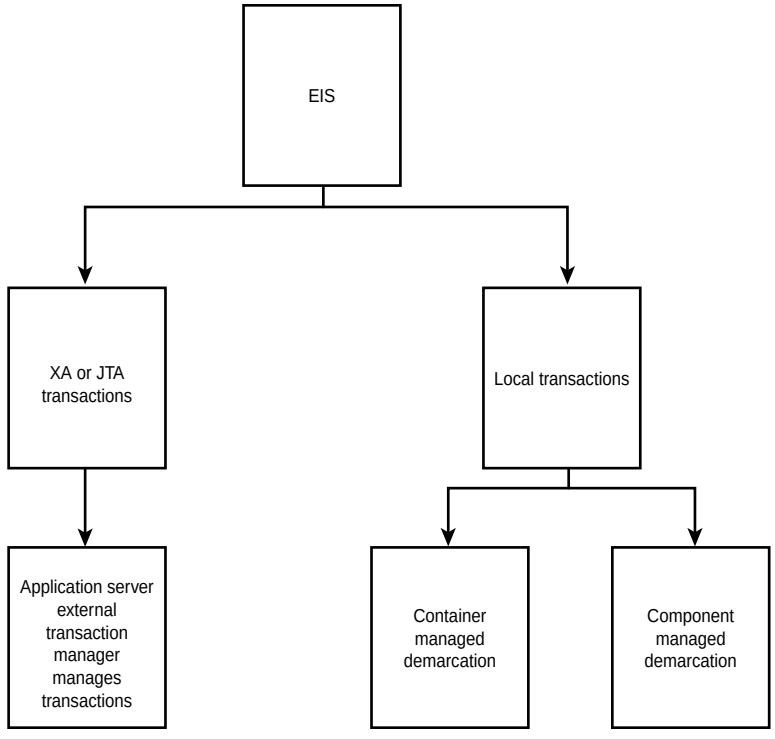
The connection contract allows the application server to pool connections to the EIS, and allows application components to connect to the EIS. Typically, when connecting to an EIS, you write code that performs a lookup in the JNDI namespace for a connection factory, which you use to gain a connection to an underlying EIS. The connection factory, on receiving a request for a connection, delegates responsibility to a connection manager on the application server. The connection manager is then responsible for checking the connection pool for a connection and, together with the application server, providing a connection instance that you can use to access the underlying EIS. Because the application server pools connections, multiple clients can connect to multiple EISs. For example, the application could manage the connections between multiple servlet instances and an ERP system and a legacy database system.

The Transaction Management Contract

The transaction management contract allows an application server to manage transactions across multiple EISs if appropriate. Figure 19.4 shows the different types of transactions the Connector architecture recognizes, and it also shows how these transactions are managed.

FIGURE 19.4

J2EE Connector architecture transaction management.



As you can see, the architecture defines two main categories of transactions—XA or JTA transactions and local transactions. The former category includes those transactions that a transaction manager on the application server manages on behalf of the EIS. In this scenario, the application server provides all runtime support for transactions, so you do not need to concern yourself with these types of transactions. The latter category, local transactions, includes transactions that the EIS manages itself. In this instance, the server-side must perform some form of transaction demarcation. The server-side can perform this demarcation in one of two ways:

- Container managed
- Component managed

Container-managed demarcation is where the component container performs demarcation on behalf of the component. For example, the EJB specification requires an EJB container to support the container-managed transactions demarcation model. Therefore, an EJB container can manage demarcation so that you do not have to explicitly begin, commit, or rollback transactions. Unlike container-managed demarcation, component-managed demarcation requires you to manage the demarcation of a transaction. You will learn how to do this in the second of the example applications in today's lesson. With regard to J2EE application components, EJB containers must support component-managed transaction demarcation, and so must Web containers that house JSPs and servlets.

For a guide to transaction management and how this affects the scalability and performance of your application, please refer to Day 8, "Transactions and Persistence."

The Security Contract

An EIS will often hold information that might be sensitive (credit card details) or mission critical (prospects details in a call center). It is vitally important for an organization to know that this information is only available to authorized persons. The J2EE Connector architecture security contract stipulates how this security is maintained when J2EE components connect to EISs. The contract aims to achieve this by extending the J2EE security model to EIS integration through the Connector architecture. One of the key goals of the contract specification is to keep security mechanism neutral. That is, the security contract is flexible enough to support a wide-range of security technologies and EISs. The two principal ways it achieves this are by not stating

- A mandatory, specific security policy
- A mandatory, specific security technology

For example, the Connector architecture specification identifies the two most commonly used authentication (identifying a user) mechanisms—basic user-password that is specific to an EIS and Kerberos Version 5. Although the specification identifies these mechanisms, it does not stipulate that a given application server or EIS should support them. Likewise, the specification provides a great deal of flexibility with regard to authorizing users. In this instance, checking whether a given user is allowed access to a particular resource can be performed either by the EIS or the application server (you will see this in the code examples in the "Using the Common Client Interface" section of today's lesson).

The whole process of signing on to an EIS from an application can occur in two ways. The first is container managed. This is where the application server takes responsibility for signing onto an EIS. Alternatively, the sign on process can be component managed. In this instance, you are responsible for writing the code that provides the authentication and authorization credentials required for sign-on.

Using the Common Client Interface

Up to now, today's lesson has concentrated on the system contract between an EIS and an application server. This information is important to assist your overall understanding of how Java applications interact with legacy and non-Java systems. But, as an application developer, you are probably eager to start writing code. This section of today's lesson looks at the relationship between an application component and a resource adapter. Specifically, you will learn how to code against the Common Client Interface (CCI) API.

A resource adapter provides a client API that you can code against. This client API may be an implementation of the CCI API, but the Connector architecture specification allows a resource adapter to support a client API that is specific to its underlying EIS. For example, an EIS that is a relational database might support the JDBC API. Even if the resource adapter does not provide a CCI implementation, a third-party can provide an implementation that sits above the resource adapter's EIS specific client API. Because of the wide variety of EISs and the APIs they may implement, today's lesson focuses on the CCI API. Always check with your resource adapter to determine the APIs it implements.

The CCI API provides a standard client API for you to code against. When you code against this API, you work with an abstraction of EIS functionality. This means that, like any other standard API, you only have to learn the one set of API calls to write code that interacts with multiple differing EISs. It also means that if you write code for one EIS and then change the underlying EIS, your code will still run against the replacement EIS.

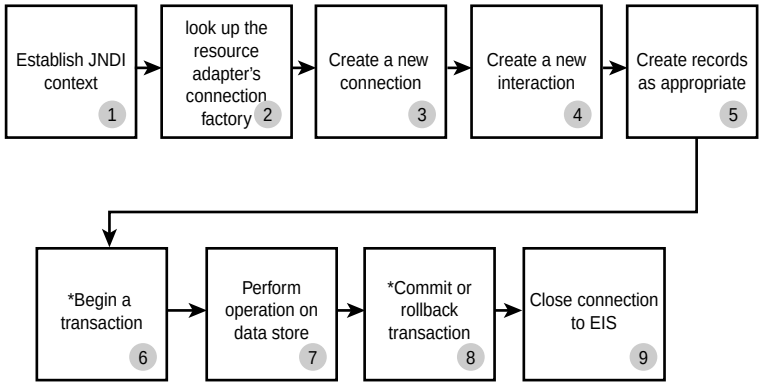
Interacting with an EIS

The process of performing an operation against an EIS involves a number of steps. Figure 19.5 shows these steps.

In a moment, you will write an application that implements these steps in code, but you will first explore them at a high-level. A resource adapter provides a `ConnectionFactory` that you use to create a `Connection` to the EIS. To locate the `ConnectionFactory`, you must first establish a JNDI `Context` for your current session and then perform a look up on the JNDI namespace to locate the `ConnectionFactory`. After you locate the `ConnectionFactory`, you can use it to create a `Connection`. As previously mentioned, the `Connection` is an application-level handle that you use to access an EIS instance.

FIGURE 19.5

The process of interacting with an EIS.



*These steps apply only if you are conducting your own transaction management.

After you have a `Connection`, you create an `Interaction`. An `Interaction` allows you to execute EIS functions. In other words, all the operations you want to perform against the EIS are done through the `Interaction`. Typically, when you execute an EIS function, you also provide the `Interaction` instance with a `Record`. The `Record` holds either the input to or the output from an EIS function. For example, you can use a `Record` instance to pass parameters to the EIS function, or you can use a `Record` instance to hold the information the EIS function returns.

If you are handling your own transaction management, you must begin a transaction before performing operations against the EIS. If you are not handling transaction management, you proceed to perform operations against the EIS. After you have performed the desired operations against the EIS, if you are managing transactions, you commit or rollback the transaction. Finally, you close the connection to the EIS.

Installing a Resource Adapter

To run the example applications in this lesson, you must install appropriate resource adapters and ensure that you have access to a corresponding EIS. To ensure that you can actually run and test the examples in today's lesson, both of the examples use the Cloudscape database, which you installed in Day 8, "Transactions and Persistence." With regard to resource adapters, you will require two resource adapters—`cciblackbox_tx` and `cciblackbox_xa`. Both of these resource adapters are samples supplied with the J2EE reference implementation.

Resource adapters come packaged in Resource adapter ARrchive (RAR) files. When you downloaded and installed the J2EE SDK, you would have also installed the resource adapters for today's lesson. You can find these resource adapters in the `lib/connector`

directory under the J2EE installation directory. If you do not have the resource adapters, you can download them by downloading the J2EE Connector architecture sample source and binary code adapters, which is available from Sun Microsystems at `http://java.sun.com/j2ee/sdk_1.3/`.

> RAR is also a recursive acronym for Roshall Archive a compressed file format. The format is named after its creator, Eugene Roshall, and you can discover more about it at `http://www.rarsoft.com`.

To install a resource adapter, start the J2EE server.

Use the `deploytool` to deploy the resource adapter. You must use the `deployConnector` switch and pass the location of the resource adapter and the server name. To do this on a Windows platform, type the following at the command prompt:

```
deploytool -deployConnector
➥%J2EE_HOME%\lib\connector\cciblackbox-tx.rar localhost
```

On a Unix platform, use the following command:

```
deploytool -deployConnector
➥$J2EE_HOME/lib/connector/cciblackbox-tx.rar localhost
```

To complete the installation, you must associate a connection factory with the deployed CCI adapter. You must use the `j2eeadmin` tool with the `addConnectorFactory` switch, passing two arguments—a JNDI name for the connection factory and the name of the resource adapter. To do this on a Windows platform, type the following at the command prompt:

```
j2eeadmin -addConnectorFactory eis/CciBlackBoxTx cciblackbox-tx.rar
```

On a Unix platform, use the following:

```
j2eeadmin -addConnectorFactory eis/CciBlackBoxTx cciblackbox-tx.rar
```

That's it, you have installed the first resource adapter. Repeat the process for the second resource adapter, `cciblackbox-ax.rar`. Finally, verify that you have correctly deployed the resource adapters by using the `listConnectors` switch of the `deploytool`. On both Windows and Unix platforms, type the following at the command line:

```
deployTool -listConnectors localhost
```

Creating a First CCI Application

This first application shows you how to write code that uses the CCI API to connect to an EIS. The application component that you will write in this instance is a Session bean;

however, you could use any other J2EE component. The bean will contain the business logic that allows you to connect to an EIS and, using a stored procedure, insert an entry into a `books` table held by the Cloudscape database.

> If you are unfamiliar with stored procedures, they are methods that perform business logic, and are stored within a database. Typically, stored procedures are written as standard SQL statements. However, some databases can use stored procedures written in Java; Cloudscape is such a database.

In this first example, you are using an EJB that uses container-managed persistence. This ensures that access to the EIS does not occur outside the context of a transaction. As such, you do not have to explicitly handle transactions as shown in steps 6 and 8 of Figure 19.5. The second example in today's lesson shows you how to manage your own transactions.

To start coding this example, you define the bean's home interface. You can see that this interface is no different from any other home interface:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface BookManagerHome extends EJBHome {
    BookManager create() throws RemoteException, CreateException;
}
```

Like the home interface, the remote interface is no different to any other remote interface. As you can see, the interface defines the `insertBook()` method that the client will use to insert a book into the EIS database:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface BookManager extends EJBObject {
    public void insertBook(String name, double price) throws RemoteException;
}
```

After you create the interfaces, you can start creating the EJB class. The first thing to note about this class is that you import two additional packages and an additional class:

- `javax.resource.cci` The package containing the CCI API interfaces
- `javax.resource.ResourceException` The root exception of the Connector architecture's exception hierarchy
- `com.sun.connector.cciblackbox`

The code itself declares four fields, which today's lesson discusses as you use them:

```
public class BookManagerEJB implements SessionBean {
    private SessionContext sc;
    private String user;
    private String pass;
    private ConnectionFactory cf;
```

In this example, there is no implementation of the `ejbCreate()`, `ejbRemove()`, `ejbActivate()`, and `ejbPassivate()` methods. Listing 19.1 at the end of this section shows their empty bodies. However, you must provide a body for the `setSessionContext()` method:

```
public void setSessionContext(SessionContext sc) {
    this.sc=sc;
    try {
        Context ic = new InitialContext();
        user = (String) ic.lookup("java:comp/env/user");
        pass = (String) ic.lookup("java:comp/env/password");
        cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
    }
    catch (NamingException ne) {
        System.err.println(ne.getMessage());
    }
}
```

The `try-catch` construct contains the code that uses the CCI API. The code starts by establishing a JNDI context (step 1 of Figure 19.5) using code with which you are familiar. You then use the JNDI context to perform three lookups (step 2 of Figure 19.5). The first two simply obtain the username and password from environment properties. The third obtains a reference to the connection factory for the resource adapter. Finally, the `lookup()` method throws a `NamingException`, so you catch this.

After you have implemented the `setSessionContext()` method, you can write the methods that contain the business logic. In this example, there is only one such method and it accepts a book name (`String`) and a book price (`double`), that it inserts into the EIS:

```
public void insertBook(String name, double price) {
```

A `try-catch` construct encapsulates the method body, because many of the methods you call might throw `ResourceExceptions` or exceptions that extend `ResourceException`. The construct begins by creating a new connection to the EIS (step 3 of Figure 19.5). To do this, you must first create a `ConnectionSpec` object that you will use to pass the username and password to the connection factory. `ConnectionSpec` is an interface, so you create an object by using the concrete class `CciConnectionSpec`:

```
ConnectionSpec cs = new CciConnectionSpec(user, pass);
```

After you create the ConnectionSpec object, you can get a Connection object by using the getConnection() method of the ConnectionFactory. This method is overloaded and, as such, has two versions, both of which return a Connection object. The first version accepts no parameters, and you should only use this if you want the EIS to manage the sign-on process. The second version, which you will now use, accepts a single parameter—a ConnectionSpec object. The method throws a ResourceException.

```
Connection c = cf.getConnection(cs);
```

To actually perform operations against the EIS, you need an Interaction object (step 4 of Figure 19.5). To get an Interaction object, you call the createInteraction() method of the Connection object. Note that the methods throws a ResourceException:

```
Interaction i = c.createInteraction();
```

To use the Interaction object, you must create an InteractionSpec object. This object allows the Interaction object to execute functions on the underlying EIS. The InteractionSpec interface in the CCI API exposes three fields and no methods. An implementation of the interface does not have to support these standard fields if they do not apply to the underlying EIS. In addition, the implementation can provide any additional fields that apply to the EIS. The specification for the interface states that the implementation must provide accessor methods (get and set) for any fields that it does support. The implementation of this interface that comes with cciblackbox provides accessor methods for three fields:

- functionName
- schema
- catalog

For the sake of completeness, you set all three of these fields in the example you are currently writing. Note, however, that the catalog is set to null, because Cloudscape does not require a catalog name. The INSERTBOOK function name refers to the stored procedure that you will execute in a moment. You will learn more about stored procedures a little later in this section.

```
CciInteractionSpec iSpec = new CciInteractionSpec();
iSpec.setFunctionName("INSERTBOOK");
iSpec.setSchema(user);
iSpec.setCatalog(null);
```

You are now ready to start creating records that hold the input to or the output from an EIS function. To create records, you must first create a record factory. The getRecordFactory() method of the ConnectionFactory object creates a RecordFactory. This method throws two exceptions—a ResourceException and one of

its subclasses, `NotSupportedException`—thrown when a resource adapter or application server does not support the operation.

```
RecordFactory rf = cf.getRecordFactory();
```

The `RecordFactory` creates two types of records—`MappedRecord` and `IndexedRecord`. A `MappedRecord` is a record that you use to hold a key-value representation of the record elements. You should find this type of record familiar because it has a super interface of `java.util.Map` (`HashMap` and `Hashtable` also implement this interface). In contrast to the `MappedRecord`, the `IndexedRecord` is a record that represents record elements as an ordered collection. This type of record allows you to access items in the collection by index, and it also allows you to search for elements within the collection. You should also find using this type of record familiar because it has a super interface of `java.util.List` (`Vector` and `ArrayList` also implement this interface). In the example you are writing, you use an `IndexedRecord` because Cloudscape only supports indexed records.

```
IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
```

The stored procedure that you are going to execute on the EIS accepts two parameters—a book name and a book price. You use the `IndexedRecord` object that you just created to pass these parameters to the procedure. To do this, you use the `add()` method that `IndexedRecord` inherits from `java.util.List`. The method returns a `boolean` that you assign to a variable named `flag`. In this instance, you discard the returned Boolean because you do not use it later in the code:

```
iRec.add(name);
iRec.add(new Double(price));
```

You can see that the code creates a new instance of `Double` rather than passes the price as a primitive double. The reason for creating the `Double` object is that when the program connects to the EIS, it converts Java object types into SQL equivalents, and when the EIS returns, the program converts the SQL type back to a Java type. The actual mapping of Java and SQL types is defined in the `Types` class of the `java.sql` package. To gain a complete reference to the Java-SQL mappings, refer to the J2SE API documentation.

The stored procedures used in this example are written in Java. It is not the objective of this lesson to explain stored procedures, but if you want to view their code, you can find them on the CD-ROM that accompanies this book.

Now that you have added the parameters to the record, you can execute the function on the EIS. To do this, you use the `execute()` method of the `Interaction` object. This

method is overloaded and, as such, comes in two varieties. The one you will use accepts an `Interaction` object and an input (containing parameters) `Record` object. When the function executes, the method returns a `Record` object containing the output, such as the results of a query. The second version of the method accepts an additional `Record` object that the method updates to include the output from the function execution. This version of the method returns a Boolean that has a value of `true` if the execution of the EIS function was successful.

```
i.execute(iSpec, iRec);
```

After you execute the EIS function, you can close the connection to the underlying EIS (step 9 in Figure 19.5). To do this, simply call the `close()` method of the `Connection` object:

```
c.close();
```

You have now completed the Session bean code to insert data into an EIS. Listing 19.1 shows the complete code.

LISTING 19.1 `BookManagerEJB.java`

```
import javax.ejb.*;
import javax.resource.cci.*;
import javax.resource.ResourceException;
import javax.naming.*;
import com.sun.connector.cciblackbox.*;

public class BookManagerEJB implements SessionBean {
    private SessionContext sc;
    private String user;
    private String pass;
    private ConnectionFactory cf;

    // Session Bean Methods
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext sc) {
        this.sc=sc;
        try {
            // Establish a JNDI intial context
            Context ic = new InitialContext();
            // Lookup the username and password
            user = (String) ic.lookup("java:comp/env/user");
            pass = (String) ic.lookup("java:comp/env/password");
            // Lookup the connection factory
            cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
```

LISTING 19.1 Continued

```
            }
            catch (NamingException ne) {
                System.err.println(ne.getMessage());
            }
        }

        // The business method
        public void insertBook(String name, double price) {
            try {
                // Create a ConnectionSpec object that holds username and password
                ConnectionSpec cs = new CciConnectionSpec(user, pass);

                // Get a connection to the EIS from the ConnectionFactory
                Connection c = cf.getConnection(cs);

                // Create an interaction, to invoke stored procedures
                Interaction i = c.createInteraction();

                /**
                 * Create an InteractionSpec,
                 * so as to pass properties to the interaction object
                 */
                CciInteractionSpec iSpec = new CciInteractionSpec();

                // Set the fields for this instance
                iSpec.setFunctionName("INSERTBOOK");
                iSpec.setSchema(user);
                iSpec.setCatalog(null);

                // Create a new record factory from the connection factory
                RecordFactory rf = cf.getRecordFactory();

                // Cloudscape only supports indexed records, so create one of these
                IndexedRecord iRec = rf.createIndexedRecord("InputRecord");

                // Add the name and price parameters or the record
                iRec.add(name);
                iRec.add(new Double(price));

                // Execute the stored procedure
                i.execute(iSpec, iRec);

                // Close the connection
                c.close();
            }
            catch(ResourceException re) {
                System.err.println(re.getMessage());
            }
        }
    }
```

To run the application, you must compile the EJB's source code, build the application, and write a client application. The client for this example is quite simple; it accesses the bean as you would any other bean, namely

- Create a new context
- Lookup the EJB
- Create instances of the home interface and the EJB itself

After you create an instance of the EJB, you call its `insertBook()` method passing a book name and book price:

```
System.err.println("Inserting book...");
bm.insertBook("Teach Yourself in 21 Days", 29.99);
```

That's it. To run the application, compile Listing 19.2.

LISTING 19.2 `BookManagerClient.java`

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class BookManagerClient {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("java:comp/env/ejb/BookManager");

            BookManagerHome bmh = (BookManagerHome)
➥PortableRemoteObject.narrow(objref, BookManagerHome.class);

            BookManager bm = bmh.create();

            System.err.println("Inserting book...");
            bm.insertBook("Teach Yourself in 21 Days", 29.99);

            System.err.println("Book inserted.");
        }
        catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```


Managing Transactions and Exploring Records

The previous example application was relatively simple; it used container-managed persistence and defined a single business method. This next example defines a method that retrieves data from the underlying EIS, and also requires you to manage transactions.

The EJB's home interface is identical to that in the previous example, except that you append a `2` to the interface and classnames:

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface BookManagerHome2 extends EJBHome {
    BookManager2 create() throws RemoteException, CreateException;
}
```

The EJB's remote interface is also very similar to that in the previous example. The important thing to note is that the method signature the interface defines now applies to a `listTitles()` method that accepts only a single parameter, a string:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface BookManager2 extends EJBObject {
    public void listTitles(String name) throws RemoteException;
}
```

All of the changes to the main EJB class apply to the business method, except that you must import the `java.util` package because you will use this later. The new business method has a signature of

```
public void listTitles(String name)
```

Like the previous example, a `try-catch` construct encapsulates the body of the method. Also exactly like the previous example, you create `ConnectionSpec`, `Connection`, `Interaction`, `CciInteractionSpec`, `RecordFactory`, and `IndexedRecord` objects:

```
ConnectionSpec cs = new CciConnectionSpec(user, pass);
Connection c = cf.getConnection(cs);
Interaction i = c.createInteraction();
CciInteractionSpec iSpec = new CciInteractionSpec();
iSpec.setFunctionName("LISTTITLES");
iSpec.setSchema(user);
iSpec.setCatalog(null);
RecordFactory rf = cf.getRecordFactory();
IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
```

After you create the `IndexedRecord` object, the code differs in a number of respects from the previous example. The first change is that you only add one parameter to the input record—the stored procedure only searches for book titles, which contain the following parameter value:

```
iRec.add(name);
```

Now that you have created and populated the input record, you should start a transaction so you can roll back to a convenient point in the case of an error. Earlier, today's lesson discussed transaction management, and it said that there were two ways to manage local transactions, either by:

- Container management—like the previous example
- Component management

In this example, you are implementing component transaction management. To do this, you use the methods the CCI API `LocalTransaction` interface defines. You should note that it is optional for a CCI implementation to implement this interface. To get a `LocalTransaction` instance, you call the `getLocalTransaction()` method of the `Connection` object. This method throws a `RemoteException` and, in the instance of an implementation not supporting local transactions, it throws a `NotSupportedException`.

```
LocalTransaction lTrans = c.getLocalTransaction();
```

After you obtain a `LocalTransaction` object, you call its `begin()` method to begin a transaction:

```
lTrans.begin();
```

At this point in the previous example, you invoked a stored procedure that inserted a record into the underlying EIS. In this example, you execute a stored procedure that performs a `SELECT` against the EIS. Because you are changing the business logic, the code changes in a couple respects. First, because the execution of the stored procedure might fail, you use a `try-catch` construct to encapsulate the code that executes the stored procedure. Second, you create a record to hold the output that the stored procedure returns. You can either create a `Record` and pass this as a third parameter to the `execute()` method of the `Interaction` object, or you can simply use the version of the `execute()` method that returns a `Record`:

```
Record oRec = i.execute(iSpec, iRec);
```

The `Record` object, `oRec`, contains the results of the `SELECT` statement the stored procedure executed against the underlying EIS. To iterate through the list of elements the `Record` object contains, you create an `Iterator` object. The `IndexedRecord` class inherits an `iterator()` method from `java.util.List`, and this returns an `Iterator` object. To call this method, you must cast the `Record` object you just created to an `IndexedRecord`:

```
Iterator iterator = ((IndexedRecord)oRec).iterator();
```

Now that you have an `Iterator` object, you can iterate through the data returned by the EIS and print each of the book titles. You use the `Iterator` object in the same way as you would in any other Java application:

```
while (iterator.hasNext()) {
    String title = (String)iterator.next();
    System.out.println(title);
}
```

That completes the processing undertaken within the `try` element of the `try-catch` construct. To complete the code, you must catch any exceptions that might be thrown and, in such an event, rollback the transaction by using the `rollback()` method of the `LocalTransaction`:

```
catch (ResourceException e) {
    lTrans.rollback();
}
```

You have now completed the Session bean code to retrieve data from an EIS. Listing 19.3 shows the complete code.

LISTING 19.3 `BookManagerEJB2.java`

```
import javax.ejb.*;
import javax.resource.cci.*;
import javax.resource.ResourceException;
import javax.naming.*;
import com.sun.connector.cciblackbox.*;
import java.util.*;

public class BookManagerEJB2 implements SessionBean {
    private SessionContext sc;
    private String user;
    private String pass;
    private ConnectionFactory cf;

    // Session Bean Methods
    public void ejbPassivate() {;}
    public void ejbActivate() {;}
    public void ejbCreate() {;}
    public void ejbRemove() {;}
    public void setSessionContext(SessionContext sc) {
        this.sc=sc;
        try {
            Context ic = new InitialContext();
            user = (String) ic.lookup("java:comp/env/user");
            pass = (String) ic.lookup("java:comp/env/password");
            cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
        }
        catch (NamingException ne) {
            System.err.println(ne.getMessage());
        }
    }
```

LISTING 19.3 Continued

```
        // The business method
        public void listTitles(String name) {
            try {
                ConnectionSpec cs = new CciConnectionSpec(user, pass);
                Connection c = cf.getConnection(cs);
                Interaction i = c.createInteraction();
                CciInteractionSpec iSpec = new CciInteractionSpec();
                iSpec.setFunctionName("LISTTITLES");
                iSpec.setSchema(user);
                iSpec.setCatalog(null);
                RecordFactory rf = cf.getRecordFactory();
                IndexedRecord iRec = rf.createIndexedRecord("InputRecord");

                // Add the name parameter to the record
                iRec.add(name);

                // Obtain a reference to a transaction context
                LocalTransaction lTrans = c.getLocalTransaction();

                // Start the transaction
                lTrans.begin();

    /**
                 * Execute the stored procedure,
                 * passing the InteractionSpec and the record
                 */
                try {
                    Record oRec = i.execute(iSpec, iRec);
                    // Get an iterator. Cast the Record to an IndexedRecord.
                    Iterator iterator = ((IndexedRecord)oRec).iterator();
                    // Iterate through the records and print names
                    while (iterator.hasNext()) {
                        String title = (String)iterator.next();
                        System.out.println(title);
                    }
                }
                catch (ResourceException e) {
                    // Rollback in the event of an error
                    lTrans.rollback();
                }

                c.close();
            }
            catch(ResourceException re) {
                System.err.println(re.getMessage());
            }
        }
    }
```

Finally, to run the application, you must compile the EJB's source code, build the application, and write a client application. The client for this example is the same as that for the previous example, except that it calls the EJB's `listTitles()` method.

That's it. To run the application, compile the code in Listing 19.4.

LISTING 19.4 `BookManagerClient2.java`

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class BookManagerClient2 {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("java:comp/env/ejb/BookManager2");
            BookManagerHome2 bmh = (BookManagerHome2)
➥PortableRemoteObject.narrow(objref, BookManagerHome2.class);
            BookManager2 bm = bmh.create();
            System.out.println("Retrieving titles...");
            bm.listTitles("Sams");
            System.out.println("Completed Retrieval.");
        }
        catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Introducing Other Connectivity Technologies

So far in today's lesson, you have learned about the J2EE Connector architecture, how it allows J2EE application components to interact with EISs, and how to code against these EISs. This is just one element of the Java technologies that allows you to connect to legacy systems and non-Java systems. There are other Java technologies that allow you to integrate Java applications with non-Java applications. The remainder of today's lesson provides an overview of three of these Java technologies:

- Java IDL
- RMI over IIOP
- Java Native Interface (JNI)

The first two technologies allow you to create and interact with objects that comply with the Common Object Request Broker Architecture (CORBA). If you don't know about CORBA, don't worry because the next section of today's lesson provides a brief overview of it. The most important aspect of these Java technologies is that they allow you to write code that interacts with either local or remote objects that are written in languages other than Java.

The third technology, JNI, also allows you to write code that interacts with code written in languages other than Java. Unlike the previous technologies, JNI allows you to interact with applications and libraries that are written in other languages, rather than only CORBA objects written in other languages. For example, you can write code that uses JNI that utilizes C libraries or C++ classes. This means that you can call non-Java functions to provide services that are unavailable from Java. The integration that JNI allows works in both directions—both the Java and non-Java sides can create, update, and access Java objects.

After today's lesson introduces these technologies, it will provide you with a brief evaluation of these technologies to help you decide which best suit your application's needs.

Introducing CORBA

The Object Management Group (OMG) defines the Common Object Request Broker Architecture (CORBA), an architecture that allows you to build distributed objects and services. The architecture is independent of any particular language implementation or system architecture. Thus, you can write remote objects in one language, say C++, and then consume them from a client object written in yet another language, such as Java. Because the CORBA standard allows communication between seemingly disparate languages, applications, and systems, it is quite extensive. However, to understand how CORBA works, there are four main aspects of the architecture that you should appreciate:

- Interface Definition Language (IDL)
- Object Request Broker (ORB)
- The Naming Service
- Inter-ORB communication

The OMG has a Web site dedicated to CORBA, that you can access at `http://www.corba.org/`.

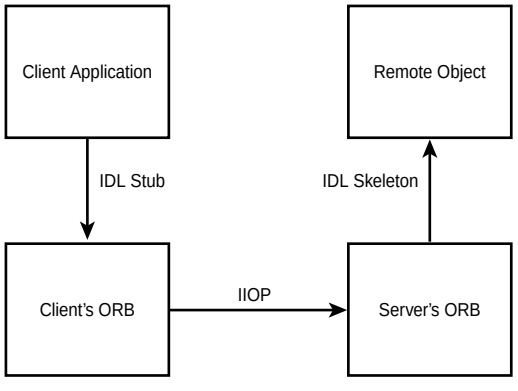
CORBA-compliant remote objects expose interfaces that you define in IDL. After you have written an IDL interface, you compile it to produce a client stub and an object skeleton. It is through the stub and skeleton that clients and objects communicate. The OMG provides a number of standard mappings that map CORBA IDL to other programming languages. Examples of these languages include

- Java
- Python
- Smalltalk
- COBOL
- C++

Java IDL, which you will learn about in the next section of this lesson, is not a mapping defined by the OMG but, rather, Sun Microsystems's implementation of the standard mapping.

All communication among objects and clients occurs through the ORB. An ORB runs on both the client and the server. Figure 19.6 shows the roles of the ORBs in client-server communication. You can see that a client application makes a request on a stub that exposes the methods of the remote object. The client ORB forwards that request to the remote ORB and, in turn, this ORB forwards the request through the skeleton to the remote object.

FIGURE 19.6
Client interacting with CORBA object.



CORBA defines a number of transport protocols that allow distributed ORBs to communicate. The most popular of these is the Internet Inter-ORB Protocol (IIOP), which is

based on TCP/IP. Although you will learn more about this protocol later in today's lesson, it is very unlikely that you will have to work with it at a low-level.

The CORBA Naming Service allows you to register an instance of a class, so that a client can look up this instance and gain a reference to it. You will learn more about how this works later in today's lesson in the "Using RMI over IIOP" section.

Introducing Java IDL

Previously, you learned that OMG provides an IDL mapping for each language that supports CORBA. You also learned that Sun's implementation of the OMG Java mapping is known as Java IDL, or to be more precise, Java IDL supports the OMG IDL mapping for Java. Using Java IDL, you can write, instantiate, and consume distributed objects that comply with CORBA. If you are an experienced Java programmer but not an experienced CORBA programmer, you may want to look at RMI over IIOP (the next section of this lesson introduces you to RMI over IIOP). This provides similar functionality without the need to write any IDL. In the context of J2EE, you will virtually always use RMI over IIOP when interacting with CORBA objects. Therefore, this lesson shows you how to use RMI over IIOP rather than Java IDL. If you want to discover more about Java IDL, refer to the current documentation available at Sun Microsystems' Web site.

Using RMI over IIOP

Remote Method Invocation (RMI) is a term you may have seen a number of times reading this book. You probably know that EJBs use RMI, but you may still wonder what exactly RMI is. Well, it is a Java-specific distributed object system that allows you to create and use remote objects. For example, an RMI object running on one server can get a reference to another RMI object running on another server. After it has that reference, it can invoke the methods of the remote object as if it were local.

As with CORBA applications, you write remote interfaces for an object and generate stubs and skeletons. Also like CORBA, RMI allows a client and a remote object to communicate through client stubs and server skeletons. The stub exposes the methods of the remote object, and the client makes requests against the stub. These requests forward to the server and pass through the server skeleton to the remote object.

The original form of RMI used a proprietary protocol, Java Remote Method Protocol (JRMP), to allow objects to communicate. This works very effectively in a Java-only environment; but as you have seen, many enterprises environment are heterogeneous. If your environment houses objects that are written in languages other than Java, you cannot use RMI with JRMP. One solution to this problem is to forget RMI and write all your objects as CORBA objects. However, this would lead to many problems. For example,

writing CORBA interfaces in Java IDL is involved, and how would EJBs communicate over distributed systems? Another solution is to create RMI objects but allow them to use IIOP, the CORBA protocol, as a transport mechanism. This approach is known as *RMI over IIOP*, or *RMI-IIOP*.

As you will learn when you write the code example later in this section, you can use RMI-IIOP to allow RMI objects to communicate directly with CORBA objects and, likewise, CORBA objects with RMI objects. This approach to interacting with CORBA objects requires you to make a few changes to a typical RMI object's code, but you do not have to write any IDL. Another very important use of RMI-IIOP is providing a transport mechanism for EJBs. When you create stubs and skeletons for EJBs, RMI-IIOP is used as the default transport mechanism. You can also allow EJBs to communicate with CORBA objects by using the remote method invocation compiler (`rmic`) to generate IDL on your behalf. When you do this, you can write an EJB client using an alternative CORBA language binding, such as C++. You will learn how to use `rmic` in the next section of today's lesson.

RMI over JRMP Example

This first example shows you how to use RMI in a completely Java environment. The example allows a remote user to enter their name, and the local server prepends the name with `hello` and returns the string to the client. There are three Java classes that you must write to produce an RMI application:

- An interface for the remote object
- An implementation of the interface
- A client for the remote object

Listing 19.5 shows the code for the interface for the remote object. As you can see, the interface is quite straightforward, but there are a couple of points to note. The first is that the interface must extend `java.rmi.RemoteInterface`. The second is that all methods that the interface declares must throw a `RemoteException`. This exception is the superclass of many of the exceptions that might occur during an RMI operation.

LISTING 19.5 `HelloUser.java`

```
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface HelloUser extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

You must now create the implementation of this interface. Naturally, this implementation class must implement the `HelloUser` interface, but is also must extend `java.rmi.UnicastRemoteObject`. This class provides an object with much of the basic functionality to allow an object to become remote. If you don't extend this class, you have to explicitly provide this functionality:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;

public class HelloUserImpl extends UnicastRemoteObject implements HelloUser {
```

When you wrote the interface, you declared that all the methods throw `RemoteExceptions`; this is also true of the class constructer. As a result, you must provide a constructer that throws a `RemoteException`:

```
public HelloUserImpl() throws RemoteException {
}
```

After you write the constructer, you can implement the method bodies. In this example, there is only one method—`sayHello()`. Remember that you must declare that each method implementation might throw a `RemoteException`:

```
public String sayHello(String name) throws RemoteException {
    return "Hello "+name;
}
```

The final part of the implementation class is to create a `main()` method that creates an instance of the implementation class and registers it with an RMI registry. Registering an object with the RMI registry allows remote clients to look it up over a network. After a remote client finds the object, the registry returns a reference to the client. The client can use this reference to invoke the methods of the remote object. To register an object with the RMI registry, create an instance of the implementation class and then use the `rebind()` method of the `Naming` class to register the class instance:

```
public static void main(String args[]) {
    try {
        HelloUserImpl hui = new HelloUserImpl();
        Naming.rebind("HelloUser",hui);
```

The `rebind()` method's two arguments are a name for the object (the client can then look up this name) and a remote object. If an existing binding for the name exists, it is replaced with this new binding. Alternatively, you can bind a name by using the `bind()` method. However, if the name already exists, the method throws an `AlreadyBoundException`. To explicitly unbind a name associated with a remote object, you use the `unbind()` method. This method takes one parameter—the name to unbind. If the name is not already bound, the method throws a `NotBoundException`.

You have now written the implementation class. Listing 19.6 shows the full code for this class.

LISTING 19.6 HelloUserImpl.java

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;

public class HelloUserImpl extends UnicastRemoteObject implements HelloUser {
    // Constructer throws RemoteException
    public HelloUserImpl() throws RemoteException {
    }

    // Implement method body
    public String sayHello(String name) throws RemoteException {
        return "Hello "+name;
    }

    public static void main(String args[]) {

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        try {
            // create a HelloUser instance
            HelloUserImpl hui = new HelloUserImpl();

            //Register the account
            Naming.rebind("HelloUser",hui);
            System.out.println("Registered");
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Now that you have written the remote object, you must write a client that accesses it. The client class is like any other Java class, except that it must create an instance of a remote object rather than one that is local. To do this, you again use the `Naming` class, but this time you invoke its `lookup()` method. This method accepts a single parameter—the name of the remote object—and returns a reference to the remote object. You specify the name as a URL using the `rmi` scheme. For example, to look up a remote object called `example` on a server named `sams`, you would pass the parameter `rmi://sams/example`. In the client you are building, the code appears as follows:

```
HelloUser hu = (HelloUser)Naming.lookup("rmi://localhost/HelloUser");
```

Notice that you have to perform an explicit cast on the object the `lookup()` method returns. You must perform the cast, because the `lookup()` method always returns an object of the type `Remote`. After you have a reference to the remote object, you can invoke its methods as if it where a local object. Listing 19.7 shows the complete code for this class, including the code that invokes the `sayHello()` method of the remote object.

LISTING 19.7 `HelloUserClient.java`

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Naming;

public class HelloUserClient {
    public static void main(String args[]) {
        if (args.length!=1) {
            System.err.println("Usage: java name");
        }
        try {
            // Lookup the remote object
            HelloUser hu = (HelloUser)Naming.lookup("rmi://localhost/HelloUser");

            // invoke the method
            System.out.println(hu.sayHello(args[0]));
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```


You have now completed all the Java code this example requires. Now you must compile the Java classes you have just written and then create a skeleton and stub for the remote object. To create the skeleton and stub, you simply run the remote method invocation compiler (`rmic`) and pass it the name of the implementation class for which it should create the skeleton and stub:

```
rmic HelloUserImpl
```

After run, `rmic` creates two java class files—`ClassName_Skel.class` and `ClassName_Stub.class`. On the server, ensure that the J2EE server is running and then start the RMI registry:

- `start rmiregistry` On Win32
- `rmiregistry &` On Unix

Now you must run the `HelloWorldImpl` class so that the runtime invokes its `main()` method, which will register an instance of the class with the RMI registry. To run the class, you must pass two property name-value pairs to the Java Interpreter. The first property is `java.rmi.server.codebase`, which allows any remote process to dynamically load classes as required. If you don't set this property, you have to physically copy all of the classes to the client. The property value is a URL that points to the code base for the application. The URL can use either the HTTP scheme or the file scheme. For example,

```
java -Djava.rmi.server.codebase=http://sams.com/classes/
```

The URL in this example does not resolve; it is purely for illustrative purposes.

In this application, either use the HTTP scheme and change the path so that it relates to your machine, or use the file scheme and again change the path to suit you machine:

```
java -Djava.rmi.server.codebase=file:/C:/classes/
```

The second property you must pass to the Java interpreter is the location of a security policy file. You must provide a security file different from the default policy file, because the default policy does not allow remote class resolution, which you require to run RMI applications with dynamic class loading. This lesson does not intend to explain policy files, refer to the appropriate document to learn more about security, but the following is an example of a security policy file that allows your application to execute:

```
grant {
    // Allow everything
    permission java.security.AllPermission;
};
```

Save this file as **`default.policy`** on both the client and the server in the directory containing the classes for this application. Finally, to complete running the `HelloWorldImpl` class, execute the following command (change forward and backward slashes to suit your system):

```
java -Djava.rmi.server.codebase=file:/C:/classes/
➥-Djava.security.policy=c:\classes\default.policy HelloUserImpl <hostname>
```

The server-side is now complete. To use the remote object from a client, you must copy the client class (`HelloUserClient`) and the interface (`HelloUser`) to client machine. In common with the server, you must provide a more relaxed security policy to facilitate dynamic class loading. In this instance, use the same policy file that you used on the

server. To run the client type, use the following command, substituting *<username>* for your name and *<hostname>* for the remote server's name:

```
java -Djava.rmi.server.codebase=http://<hostname>/classes/
-Djava.security.policy=default.policy HelloUserClient <username> <hostname>
```

When the code executes, the client gains a reference to the remote objects and dynamically loads the stub from the server. The client application is then able to invoke the remote object's methods through the stub.

RMI over IIOP Example

Previously, today's lesson told you that you can use RMI over IIOP to write CORBA objects, and that EJBs used IIOP as a standard transport protocol. After you write a remote object so that it uses IIOP rather than JRMP as a transport protocol, it can communicate with CORBA objects, and they can communicate with it. At a high-level, the process to achieve this is quite straightforward:

1. Create an interface for the remote object.
2. Write a Java implementation of the interface.
3. Use `rmic` (with a special switch) to generate IDL stubs and skeletons.
4. Use the CORBA naming service rather than the RMI registry.

As you may guess, creating an object for use over IIOP requires you to make some changes to a standard RMI application. To illustrate this, today's lesson will walk through the process of converting the previous example application so that it uses IIOP rather than JRMP.

The remote object interface requires no changes, so the code in Listing 19.5 is still applicable in this application. However, the implementation of this interface and the client application both require modification. The implementation class no longer extends `UnicastRemoteObject`, but extends `javax.rmi.PortableRemoteObject` instead.

```
public class HelloUserImpl extends PortableRemoteObject implements HelloUser {
```

The only other change to this class is that you no longer use `Naming.rebind()`. Instead of this, you must create an `InitialContext` object and then use its `rebind()` method. The `InitialContext` class implements `Context`, which represents a naming context that consists of pairs of name-object bindings. The `InitialContext` class has three constructors:

- One creates a context based on a `hashtable` of environment information.
- The second constructs a context without initializing it.
- The third simply creates an initial context.

After you create the initial context, you call its `rebind()` method to bind the `HelloUser` instance (you created this previously) and a name for that instance:

```
Context ctx = new InitialContext();
ctx.rebind("HelloUser",hui);
```

That is the extent of the modifications to the implementation class. Listing 19.8 shows the complete, modified code for the class.

LISTING 19.8 `HelloUserImpl.java`

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.rmi.*;

public class HelloUserImpl extends PortableRemoteObject implements HelloUser {
    public HelloUserImpl() throws RemoteException {
    }

    public String sayHello(String name) throws RemoteException {
        return "Hello "+name;
    }

    public static void main(String args[]) {

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        try {
            HelloUserImpl hui = new HelloUserImpl();

            // Create an initial context
            Context ctx = new InitialContext();

            //Register the instance with initial context
            ctx.rebind("HelloUser",hui);
            System.out.println("Registered");
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Like the implementation class, the client must also create an initial context rather than use `Naming.lookup()`. Previously, you used the following line of code to look up the remote object:

```
HelloUser hu = (HelloUser)Naming.lookup("rmi://localhost/HelloUser");
```

Now you replace this line with the following:

```
Context ctx = new InitialContext();
HelloUser hu = (HelloUser)PortableRemoteObject.narrow
➥(ctx.lookup("HelloUser"),HelloUser.class);
```

You can see that you do not simply cast the object returned by the lookup in this instance, but you use the `narrow()` method of the `PortableRemoteObject` class instead. This method accepts two parameters. The first is an object from which to narrow. The second is a `java.lang.Class` object to which to narrow. In this instance, the code references the `Class` object by appending `.class` to the name of the class.

That is the extent of the changes to the Java code. Listing 19.9 shows the complete, modified code for the client class.

LISTING 19.9 `HelloUserClient.java`

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;


public class HelloUserClient {
    public static void main(String args[]) {
        if (args.length!=1) {
            System.err.println("Usage: java name");
        }

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        try {
            // Create an initial context
            Context ctx = new InitialContext();

            // use narrow to narrow object to object of the given class type
            HelloUser hu =
➥(HelloUser)PortableRemoteObject.narrow(ctx.lookup("HelloUser"),HelloUser.class);

            System.out.println(hu.sayHello(args[0]));
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

As you did previously, compile the entire Java source to generate the class files. After you have done this, you need to create a client stub and a server skeleton. Like the

previous example, you use `rmic`, but in the current example, you pass `rmic` an `iiop` switch to create an IIOP skeleton and stub:

```
rmic -iiop HelloWorldImpl
```

When you execute this command, two new files are created—`_HelloWorldImpl_Tie` (the skeleton) and `_HelloWorldImpl_Stub` (the stub). You have now generated all the files you require. To run the application, you must first start the J2EE server. After you start this, you must start the Transient Name Server—previously, you started the `rmiregistry`. The Transient Name Server provides a CORBA naming service that also has JNDI access:

```
tnameserv
```

All that remains is for you to run `HelloUserClient` and `HelloUserImpl`. To do this, you must pass additional environment properties to the Java interpreter. The first is naming factory provider, which in this application provides naming functionality for the CORBA Naming Service. The second is naming provider; for more information on JNDI environment properties, see Day 4, "Introduction to EJBs." Consequently, to start execute the implementation class on the server, issue the following command (remember to change the slash orientation to suit your system, and replace *<hostname>* with your server's name):

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
➥-Djava.naming.provider.url=iiop://<hostname>:900
➥-Djava.rmi.server.codebase=http://<hostname>/rmi/
➥-Djava.security.policy=m:\rmi\default.policy HelloWorldImpl <hostname>
```

This example uses port 900, but if you are using Unix (including Solaris and Linux), you must specify a port number greater than 1024 because the lower ports are privileged ports for reserved purposes.

To start the client, issue the following command substituting *<username>* for your name, and *<hostname>* for your hostname:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
➥-Djava.naming.provider.url=iiop://<hostname>:900 HelloWorldClient <username>
```

Introducing JNI

The Java Native Interface (JNI) allows you to write code that utilizes code written in programming languages other than Java. If you consider the maxim "Write once, run

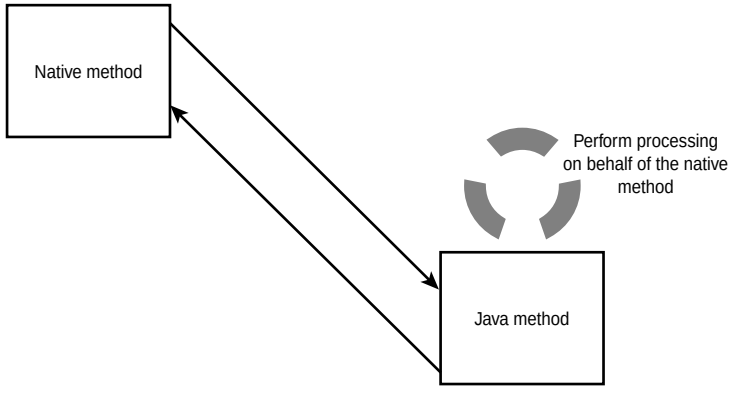
anywhere," you may wonder why you might want to use non-Java code. There are a number of scenarios where it is preferable or even wise to use non-Java code:

- When you require functionality not supported by the standard Java class library. For example, you may need to access parts of the Win32 API.
- When you want to reuse a library or application written in another programming language, so you don't have to rewrite these libraries or applications. For example, many organizations posses large legacy C libraries.
- When you need to utilize a lower-level programming language, such as assembly, to provide support for a time-critical section of code.

JNI allows you to write code that supports these types of scenario. You can use JNI to declare native methods, but implement the methods bodies in native code, such as C or C++. These native methods can use Java objects and methods in the same way that Java code uses them. Specifically, both native methods and Java methods can create Java objects, use them, update them, and then share them interchangeably. Like a native method using a Java object, a native method can invoke Java methods. For example, Figure 19.7 shows a native method invoking a Java method and passing parameters to it. The Java method performs some processing of the parameters, returns the result to the native method, and the native method then uses this result for some further purpose.

FIGURE 19.7

Native method invoking a Java method.



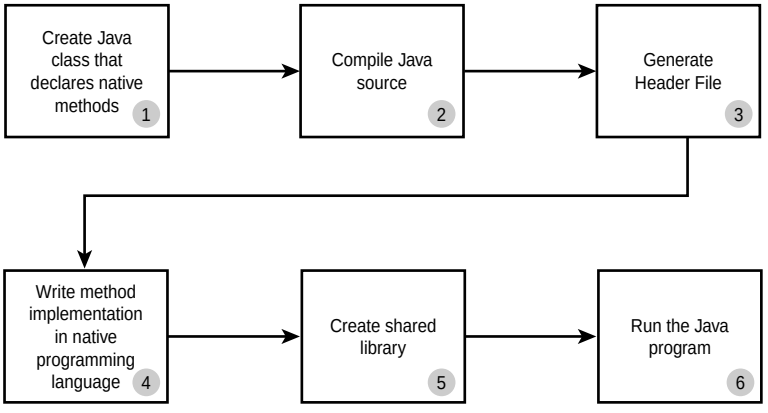
Beyond these interoperability aspects, JNI allows you to perform further tasks including the following:

- Throwing and catching exceptions from a native method, and then the Java application handles them.
- Through the Invocation API, you can embed the JVM into native applications.

- Special JNI functions allow native methods to load Java classes and obtain class information.
- Native methods can use JNI to perform runtime type checking.

A full exploration of JNI is beyond the scope of this book; however, today's lesson aims to provide you with enough information to make an informed choice of whether you might find JNI of use. For further information on the features of JNI, please refer to Sun Microsystems' JNI specification. To complete your introduction to JNI, today's lesson shows you how to write a simple JNI application—namely, `Hello World`. Figure 19.8 shows the six steps you will follow to write the application.

FIGURE 19.8
Creating a JNI application.



As Figure 19.8 shows, the first step is to create a Java class that declares the native method. To do this, declare your class and provide the method signature for the native method. Notice that the code uses the `native` modifier to indicate to the compiler that the method implementation is a programming language other than Java:

```
class HelloWorld {
    public native void displayHelloWorld();
```

Later, when you write the native code, you will compile it into a shared library. To allow the runtime to load this library into the Java class, you use the `loadLibrary()` method of the `System` class in the context of a `static` initializer. The method takes a single parameter—a string that is the name of the library to load. You only need to pass the root of the library name, because the method modifies the name to suit the current platform. For example, it will use `hello.dll` on Windows or `libhello.so` on Solaris:

```
static {
    System.loadLibrary("hello");
}
```

Finally, you must write the `main()` method in the same way as you would for any other Java class. Listing 19.10 shows the completed code for this class. After you have written it, compile it as if it were any other Java class (`javac HelloWorld.java`).

LISTING 19.10 `HelloWorld.java`

```
class HelloWorld {
    public native void displayHelloWorld();
    static {
        System.loadLibrary("hello");
    }

    public static void main (String args[]) {
        HelloWorld hw=new HelloWorld();
        hw.displayHelloWorld();
    }
}
```

Step 3 of the process is to generate the header file for the native method. Creating the header file is simple; at the command line, ensure that you are in the directory of the `HelloWorld` class file and then use `javah` by issuing the following command:

```
javah –jni HelloWorld
```

The `jni` switch instructs `javah` that it should output a header file for use with JNI. This is the default behavior in Java 1.2 and later, but earlier versions need the switch so that they do not output header files for use with the older JDK 1.0 native interface.

After you run `javah`, it creates a header file (`HelloWorld.h`) in the current directory. You don't need to look at this file, but if you want to open it up in a text editor, the native method signature is as follows:

```
Java_HelloWorld_displayHelloWorld (JNIEnv *, jobject);
```

All generated method signatures follow the same format:

```
java_packagename-classname-methodname
```

In the application you are currently writing, there is no package name, so it is omitted from the method signature. In this application, you write the native method implementation after writing the Java class file, but in reality, you may already have native methods you want to use. In this instance, you must ensure that the native method signature (in the method implementation in native code) matches the method signature in the generated header file.

Now you must write the native implementation of the method. The code in this application is written in C, but don't panic because the code is very simple to follow. The code (shown in Listing 19.11) begins by including three header files—`jni.h`, `HelloWorld.h`, and `stdio.h`. All native implementations must include `jni.h` because this provides information that allows the native language to interact with the Java runtime. You include `HelloWorld.h` because this is the header file that you just generated. Finally, you include `stdio.h` because the `printf` function (you use this to print `hello world`) is contained within this library.

Because this is not a C tutorial, there is only one other thing to note about the code. You can see that the method accepts two parameters of the types `JNIEnv` and `jobject`. All native methods must accept these parameters. The first, `JNIEnv`, is an interface pointer that allows the native code to access any parameters your Java code passes to it. The second parameter, `jobject`, references the current object itself.

LISTING 19.11 `HelloWorldImp.c`

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Now that you have written the native code, the second-to-last step shown in Figure 19.8 is to compile the header and implementation file into a shared library by using your favorite C compiler. The shared library must have the same name that was used in `loadLibrary()` method in the Java class—namely, `hello`. The actually command for this operation depends on the C compiler that you use; the JDK doe not provide such a compiler, as you might expect.

You have written the application. To run it, simply execute the Java class file you wrote earlier:

```
java HelloWorld
```

When the code executes, the runtime loads the shared library into the Java class. The `main()` method of the Java class invokes the `displayHelloWorld()` method of the native class, which in turn prints `Hello World` to the standard output.

This example has shown you how to use JNI in a very simple situation. In reality, you may have to integrate large amounts of code using JNI, so standalone command-line applications will certainly be unsuitable for use in the J2EE arena. A viable approach to making legacy code available to J2EE components is to wrap the code using JNI (as you did in the example application). Then, export the code as an RMI remote object, as shown previously. This approach allows J2EE components to interact directly with RMI objects, thus abstracting the underlying legacy code.

Evaluation of Integration Technologies

Today's lesson has provided you with an introduction to a number of approaches to integrating legacy and non-Java code or applications into your J2EE applications. Sometimes you may find it quite clear which approach to adopt, but sometimes the approach to adopt might not be immediately obvious. The following guidelines will assist you in making that choice.

- *Connector Architecture*—This is your primary choice when you want to allows J2EE components to call on the functions an EIS provides. Typically, these EISs include ERP systems, mainframe transaction processing systems, and databases (not to the exclusion of JDBC).
- *JavaIDL and RMI-IIOP*—If you want to utilize non-Java objects and the remote system is incapable of running a stable JVM or runs existing CORBA objects, you will use CORBA. You should use Java IDL if your existing code-base makes use of Java IDL, or you are an experienced CORBA programmer wanting to use Java. In most other instances, you will find it simpler and more effective to use RMI-IIOP.
- *JNI*—You should wrap code by using JNI when that code is a local non-Java application, library, or time critical sections of code that you want to implement in a lower-level programming language. In addition, you can wrap remote non-Java code by using JNI, and then export it as RMI remote objects. Note that to adopt this approach, you must have a JVM on both the client and the server.

Summary

In today's lesson, you learned about legacy and non-Java systems and how you can integrate J2EE applications with these. The lesson illustrated the J2EE Connector architecture, including the roles and contracts it defines. Building on this, you learned about the Common Client Interface and then wrote an application that utilized it.

The lesson continued by providing you with a high-level introduction to CORBA. You then learned how two Java technologies, Java IDL and RMI over IIOP, allow you to write CORBA objects and communicate with them. Finally, you learned how to use the Java Native Interface (JNI) to provide seamless interaction between Java applications and non-Java code, whether it was a lower-level language, a legacy library, or an application written in a non-Java programming language.

There were a lot of aspects to today's lesson, and this breadth meant that it wasn't possible to provide an absolutely complete reference to each of the technologies. However, the lesson has provided you with enough information on how and why you use these technologies for you to make an informed choice of which might best suit your particular application development requirements.

Q&A

Q What is the role of a resource adapter in the J2EE Connector architecture?

A A resource adapter is a software driver that acts as a bridge between an EIS and a J2EE container. The J2EE Connector architecture specification defines the relationship between the EIS and an application server through the system contract. The system contract dictates the responsibilities of both parties with regard to connection pooling, transaction management, and security. These operations are transparent to application components, which simply invoke functions on the EIS via an API exposed by the resource adapter.

Q How do I manage the demarcation of transactions when using the CCI API?

A The J2EE Connector architecture specification recognizes two categories of transaction—XA or JTA and local. The former category of transactions is managed by a transaction manager on the application server. Local transactions are managed either by the container or the component. You can demarcate component managed transactions by using the methods of the `LocalTransaction` class in a CCI API implementation.

Q I have a connection to an EIS, but I can't invoke its functions. What am I missing?

A After you establish a connection to an EIS, you must create an `Interaction` object. All EIS functions are invoked through the `Interaction` object.

Q Which Java technologies allow me to consume CORBA objects?

A Two Java technologies allow you to consume CORBA objects—Java IDL and RMI over IIOP. Java IDL is based on the OMG Java mapping for IDL. To use this approach, you write a remote object's interface in IDL. RMI over IIOP uses the Java-specific RMI technology. You write a remote object's interface and then use the RMI compiler to generate stubs and skeletons for use with IIOP, a CORBA transport protocol.

Q I have legacy code written in C that I would like to access remotely. How might I do this?

A Although C is not object-oriented, you can still wrap C code using JNI. After you wrap the code, you can export it as an RMI object, which a client can access remotely.

Exercises

Bill is having trouble piecing together all the different elements of the architecture of an e-commerce system. At the heart of the system is a J2EE server. Underlying this there is

- An extensive legacy C library that provides a number of cryptographic functions the system requires
- An ERP that is used for managing customer service

In addition, the application must automatically forward orders to two of the company's suppliers. The first provides a public interface through the use of CORBA objects, and the second through RMI.

Please help Bill by devising a suitable architecture for his e-commerce system. Create a visual representation of the architecture, ensuring that you highlight any J2EE components, the legacy and non-Java elements, and customers and suppliers. Briefly justify your choice of architecture.

WEEK 3

DAY 20

Using RPC-Style Web Services with J2EE

So far, you have seen how to use existing J2EE technologies to build multitier applications. However, the world moves on. A key area of interest at the turn of the millennium is how to integrate applications both within and between organizations. Web Services provide a flexible and powerful integration mechanism that can be used to expose existing functionality and components to other organizations or to new applications. Today and tomorrow, you will see how you can use Web Services to build bridges between J2EE application components and any other platforms that support Web Services.

Web Services are seen by many as the next wave of the Internet revolution. The vision is of a Web as rich with functionality as the current Web is with information. The challenge is to expose this functionality in a consistent and usable way.

Today, you will

- Examine the concepts underlying Web Services and how Web Services fit with J2EE
- Create a client for an RPC-style Web Service

- Implement an RPC-style Web Service
- Generate client code from Web Services Description Language (WSDL) documents and generate WSDL documents from your server implementations
- Pass complex Java types between client and service

First, you need to understand why you would use Web Services.

The aim of the last two days is to describe how to use J2EE technologies to implement and access a Web Service. This chapter will give an overview of the Web Service architecture and show how to generate and consume SOAP messages based on a WSDL interface.

Before proceeding further, please be aware that the subject of Web Services is in itself very large, and there are many books dedicated to this popular topic. Today and tomorrow are intended to give you a start into using Web Services in Java and with J2EE technologies. However, it is not possible to answer every question or to pursue every topic. If you would like to find out more about Java and Web Services after you have read through the material in this book, try the following URLs:

- Sun Java Web Services—`http://java.sun.com/webservices/`
- IBM DeveloperWorks—`http://www-106.ibm.com/developerworks/webservices/`
- Apache XML—`http://xml.apache.org/`
- Web Services Architect—`http://www.webservicesarchitect.com/`
- Web Services Portal—`http://www.webservices.org`
- ebXML home and resources—`http://www.ebxml.org`

Web Service Overview

This first section provides the underlying information and concepts required to successfully implement Web Services. Before employing Web Services, you should understand what problems they are designed to solve and the motivation behind them. This should ensure that you apply Web Services in appropriate places within your application.

What Is a Web Service?

Web Services can be seen as the next stage in the evolution of software. Procedural programming evolved into object-oriented (OO) programming to improve the modelling of

system elements and the encapsulation of data and functionality. Component-based development provides a standardized, service-rich framework in which OO functionality can be delivered and built into applications. Web Services takes advantage of common Web protocols to make component instances easily accessible both within and outside an organization.

A Web Service is essentially an application component that can be accessed using Web protocols and data encoding mechanisms, such as HTTP and XML. In some cases, this will be a third-party component hosted remotely. The difference between a Web Service and a traditional component lies not only in the protocols used to access it, but also in that the service can bring its own "live" data and "back-end" functionality with it. An example of this would be a currency conversion service. Under the component model, a currency conversion component could bring with it a file containing a fixed set of currency conversion rates that must be updated regularly. However, it would be up to you to ensure that this information is updated. On the other hand, a currency conversion service takes responsibility for this updating. Your application simply makes use of the conversion service and leaves the details of obtaining the required data and subsidiary services to those who implement and host the service.

Similarly, a Web Service may represent a courier service or a credit-card processing service. Again, you do not need to concern yourself with how the service is implemented, simply the results of using the service. There are many types of Web Services appearing that provide a sliding scale of functionality from low-level infrastructure to high-level business services.

Applications can be built from services in a similar way to building applications from components. You will combine standard services (such as credit-card authorization) with custom code to create your desired application.

As a software developer, you may write Web Services for others to use. In this case you would

1. Have a good idea for a service.
2. Implement the service being offered.
3. Describe the service being offered.
4. Publish the description.
5. Wait for customers or consumers of your Web Service.

Alternatively, you may use Web Services as part of your application as follows:

1. Discover an interesting service.
2. Retrieve the description.

3. Plug it into your application.
4. Use the service as the application executes.

This all sounds very easy, but you need a ubiquitous framework for Web Services to stop this from sliding into chaos. The key factor in delivering such a framework is the widespread agreement to use common, Web-based protocols. In the first instance, this comes down to the use of the Simple Object Access Protocol (SOAP), which is a combination of XML and HTTP. SOAP provides the transport mechanism over which Web Services communicate. Other protocols are also required to deliver the full framework and you will encounter these protocols over the course of the next two days.

Why Use Web Services?

Web Services bring similar advantages to the use of components. Using a service allows you to take advantage of another organization's expertise in, say credit card processing, without you having to become a specialist in it yourself. The service model allows you to use the most powerful and up-to-date functionality by connecting to a remote running service.

Although a service-based approach to application development is not a new concept, it has traditionally presented difficult challenges:

- Interoperability between different distribution mechanisms, such as CORBA, RMI, and DCOM.
- Application integration, including legacy systems, cross-vendor, and cross-version.
- Web-based business requires cross-organization development, high flexibility to accommodate a rapid rate of change, and safe operation through company firewalls.

Web Services can provide a consistent, cross-organization, cross-vendor framework that will speed up the integration of applications and application components. By selecting existing, widely-used standards, the Web Service framework removes many barriers to integration that existed when using other frameworks. The Web Service model is language- and platform-neutral, so developers anywhere can potentially build and consume Web Services.

Probably most important of all is the fact that all the major application, platform, and technology vendors have adopted the Web Service concept. This means that Web Services will form a large part of application development over the next few years.

Web Service Technologies and Protocols

The following are the central protocols, technologies, and standards in Web Services:

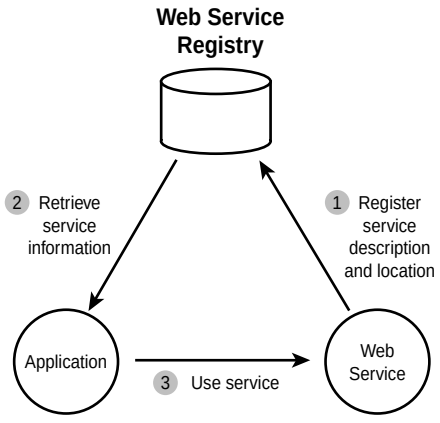
- The Simple Object Access Protocol (SOAP) combines XML and Multipurpose Internet Mail Extensions (MIME) to create an extensible packaging format. The SOAP envelope can be used to contain either RPC-style or document-centric, message-style service invocations. A SOAP message can be carried over many transport mechanisms, including HTTP, SMTP, and traditional messaging transports. Although SOAP began its life outside the World Wide Web Consortium (W3C), ongoing work on SOAP can be found at `http://www.w3.org/2002/ws/`. This includes the latest working drafts of the 1.2 specifications, as well as a link to the version 1.1 specification.
- The Web Services Description Language (WSDL) is an XML vocabulary used to describe Web Services. It defines operations, data types, and binding information. The WSDL specification can be found at `http://www.w3.org/TR/wsdl`.
- Universal Description, Discovery, and Integration (UDDI) provides a model for organizing, registering and accessing information about Web Services. The UDDI specifications can be found at `http://www.uddi.org/`.
- The Web Service Flow Language (WSFL) and Web Service Collaboration Language (WSCL) are concerned with describing the workflow between services so that their relationships can be encapsulated as part of an application. More information on WSFL can be found at `http://xml.coverpages.org/wsfl.html`.
- Electronic Business XML (ebXML) provides a framework for e-commerce that includes the inter-application workflow, and the description and discovery of services. It uses SOAP as its transport mechanism but does not directly use WSDL, UDDI, or WSFL. ebXML is a joint initiative between OASIS and the United Nations CEFACT group. The set of ebXML specifications can be found at `http://www.ebXML.org/`.

Web Service Architecture

The interaction between a Web Service-based application and the Web Service itself is shown in Figure 20.1. The overall interaction is very similar to the way that a J2EE client uses an EJB. When a Web Service is created, information about its interface and its location are stored in a registry. The Web Service consumer can then retrieve this information and use it to invoke the Web Service.

FIGURE 20.1

Interaction between Web Service, registry, and service consumer.



Some of this consumer/service interaction takes place at design and development time. The interface and service contract information will can be registered, regardless of whether the service is active or not. This information is required by the application builder to create code that uses the Web Service in their application. At runtime, the application can look up the precise location of the Web Service to locate it, very much like a traditional RPC mechanism, such as RMI.

There are several variations on this interaction. A Web Service can be used entirely dynamically in that the service description is discovered and invoked dynamically. Alternatively, the location information discovered at design time as part of the service description can be bound into the client application so that it has no need of the registry at runtime.

Similarly, the way in which an application interacts with a Web Service will depend on the service. Some services may provide an RPC-style interface based on request/response operations while others may work in a massaging style by exchanging XML-based documents. In either case, the interaction can be synchronous or asynchronous. There is nothing to stop a Web Service from offering out its services in all four combinations.

Service developers will define an interface for their service using a description mechanism such as WSDL. This can be based on an existing service implementation, or the service can be developed after the interface is defined.

Application developers will take the service description and write code based on this. In many cases, a client-side proxy will be created for the services and the application will interact with this proxy. However, the precise details of this are left to the client-side developer.

The service implementations will take a variety of forms. On the server-side, an adapter and router will be required to accept inbound SOAP messages and dispatch them to the appropriate service implementation. This performs the role of the Object Request Broker (ORB) in CORBA and RMI or of the Service Control Manager (SCM) under DCOM.

The services being invoked can be of varying granularity. Web Service mechanisms can be used as a convenient way to integrate existing, fine-grained components. Alternatively, the Web Service being accessed can represent a whole application, such as an ERP system.

Although there is much about the Web Service paradigm that will seem familiar to you, the use of Web Services, especially third-party Web Services, does bring some extra considerations for developers:

- The fact that the service is hosted elsewhere will impact testing, security, availability, and scalability. There will be a need for Service-Level Agreements (SLAs) to be defined for all services used.
- The providers of an external service will have to be paid somehow. There will be associated authentication requirements so that use of the service can be tracked by the providers.

Web Services for J2EE

At the time of writing, Web Services are not an integral part of J2EE. However, the overall intention and roadmap has been laid out. This section examines how Web Services fit with the J2EE model and how they can be used with J2EE components.

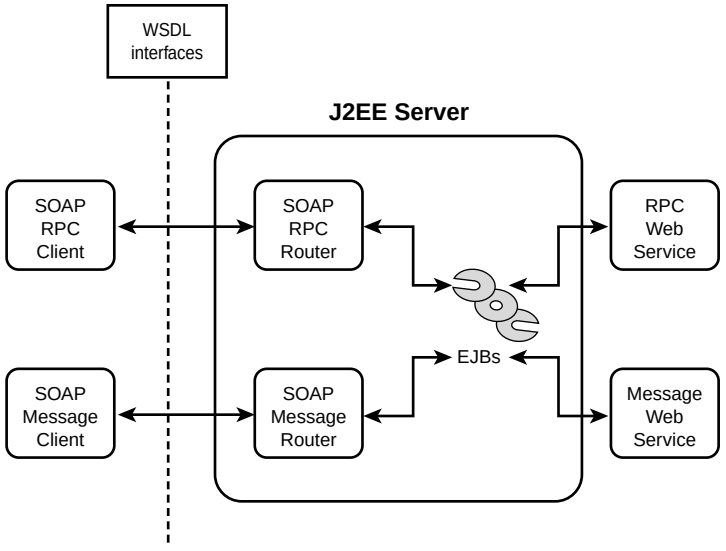
J2EE Web Service Architecture

J2EE will be both a provider and consumer of Web Services. Figure 20.2 shows the overall architecture, with business logic being provided by EJBs (although other classes could be used). The functionality offered by the business components will be described by a WSDL document (or similar), and this can then be used to build clients that use this functionality.

SOAP RPC calls will be handled by a router component based around a servlet. This will dispatch calls to the associated EJB or other component. The router that handles document-centric SOAP messages will also be servlet-based. In either case, the precise nature of the servlet will depend on the type of underlying transport over which the messages are sent.

The J2EE business components may themselves use other Web Services to help them deliver business functionality. In this case, the components will take advantage of the client-side Web Service APIs to call out to these Web Services.

FIGURE 20.2
Overall J2EE Web Service architecture.



The Web Service runtime will consist of a variety of filters, providers, and helpers that will be used in combination with the routing servlets and basic, low-level APIs. These helpers will deliver additional value on top of the basic APIs, such as ebXML quality of service guarantees.

Tools and Technologies

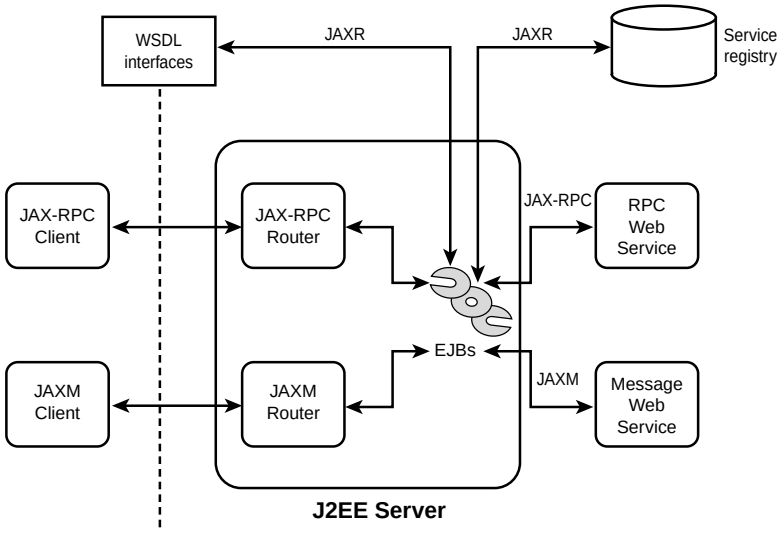
There are a number of JSRs that are in progress in the Java Community Process (JCP) to define Web Service APIs for Java. These include the following:

- JSR101, Java APIs for XML-based RPC (JAX-RPC), provides APIs for invoking RPC-based Web Services over SOAP. It defines how interactions should take place and provides the basis for automated tools to produce stubs and skeletons. It also specifies type mapping and marshalling requirements between Java and SOAP/WSDL.
- JSR067, Java APIs for XML Messaging (JAXM), defines APIs for creating document-centric SOAP messages that can be exchanged either synchronously or asynchronously. Vendors can provide messaging profiles on top of this that offer value-added services, such as ebXML.
- JSR093, the Java API for XML Registries (JAXR), defines a two-tier API for accessing registry information stored in XML format. This is targeted at Web Service-related registries, such as UDDI registries and ebXML registry/repositories, as well as other generic XML registries.

The contents and status of these JSRs are available through the JCP Web site at `http://www.jcp.org/`.

The role that each of these APIs plays in the J2EE Web Service architecture is shown in Figure 20.3. All of these APIs are intended for inclusion in J2EE 1.4 (as defined in JSR151). In the interim, they will be delivered as part of the JAX Pack, along with other Java APIs for the manipulation of XML. The first JAX Pack was delivered in Fall 2001.

FIGURE 20.3
J2EE Web Service APIs.



Until the finalization and release of J2EE 1.4, there are various sources of Java-based Web Service functionality:

- The Apache Software Foundation provides the Axis toolkit for the creation and use of SOAP-based Web Services that can be deployed in most servlet containers. Axis is tracking the JAX-RPC JSR as well as the progress of SOAP 1.2. The predecessor to Axis was Apache's SOAP toolkit 2.2. The Axis toolkit can be found at `http://xml.apache.org/axis`.
- IBM provide their Web Services Toolkit (WSTK) through their Alphaworks developer site. The WSTK provides a set of tools and APIs on which to build Web Services. The WSTK integrates with the Apache Tomcat servlet engine and IBM's WebSphere application server. IBM's Web Service Toolkit can be found at `http://alphaworks.ibm.com/tech/webservicestoolkit`.

- As the various JSRs reach maturity, they will be obliged to release a reference implementation (RI) of their functionality. These various RIs are available for individual download from Sun and are also bundled as part of the JAX Pack, also available from Sun. You can download the latest versions of these implementations from `http://java.sun.com/webservices`.
- Some Web Service functionality is available in shipping J2EE application servers. An example of this is the BEA WebLogic server 6.1 that provided Web Service functionality that pre-dates the final outcomes of any of the related JSRs. Due to the nature of the JCP, most vendors are able to track the progress of the JSRs and deliver functionality early to their customers.

If you want to investigate or use Web Service functionality in your applications, the appropriate choice will depend on the style and robustness you require.

At the time of writing, the Web Service standards and their Java APIs were still works in progress. Hence, the primary vehicle used in subsequent sections for creating and using RPC-style Web Services is Apache's Axis toolkit. This shows many indications that it is tracking the JAX-RPC JSR and so should have similar features and tools to the eventual reference implementation.

Integrating Web Services with Existing J2EE Components

Most development projects involve using or adapting existing functionality. Projects based on Web Services will be no different. In fact, a project can be specifically focused at exposing existing functionality to clients in the form of Web Services. So, how do you expose existing J2EE functionality as Web Services?

For a traditional J2EE application, business logic is contained in EJBs. This functionality is usually made available to client applications through servlets and JSPs. If the clients are Web browsers, these Web components will generate and consume HTML. Similarly, if the clients are mobile applications, the Web components may generate and consume Wireless Markup Language (WML). However, these WML Web components share the same business logic—they just provide a different front-end or channel for it. Web Service clients are no different in that respect from HTML or WML clients. The SOAP router servlet, together with helper classes, acts as a server-side wrapper for the business logic, delivering it to Web Service clients. This situation is shown in Figure 20.4.

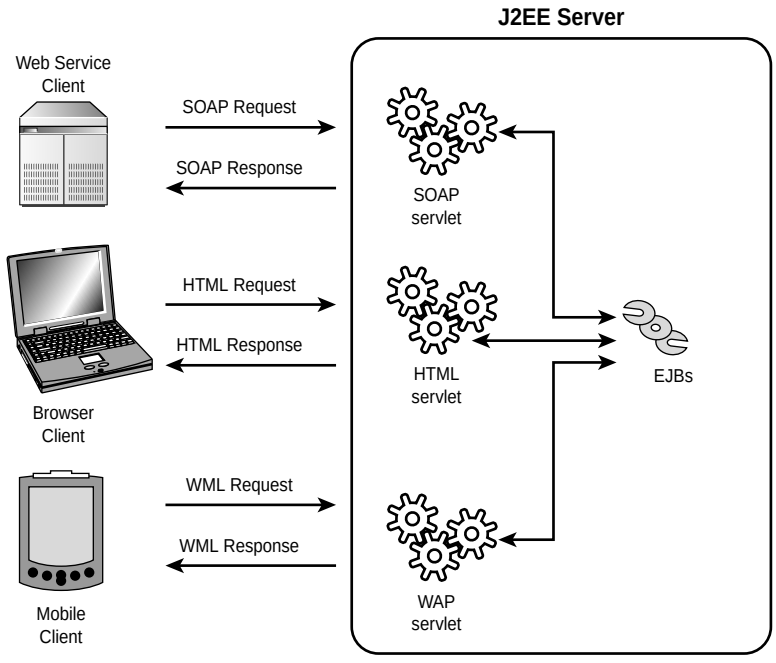


FIGURE 20.4

Web Services are just another channel through which to access business functionality.

The other type of J2EE component in which application logic can be held is a servlet or JSP. You may ask how you would wrap this functionality for use as a Web Service. Well, the issue here is that many of the Web components in question are already acting as channels to some form of client (as shown in Figure 20.4). Consequently, wrapping them makes no sense. What you should do is create a replacement for such a Web component that is targeted at Web Service clients rather than Web browsers. If your Web components are well designed, you should be able to reuse the JavaBeans, servlet filters, and helper classes (even servlets/JSPs that they use) as part of your Web Service implementation. If you already have servlets or JSPs that generate XML, you might be able to migrate them to meet your Web Service needs or transform the generated XML as part of the solution.

Using an RPC-style SOAP-Based Web Service

SOAP grew out of an effort to create an XML-based method invocation mechanism for distributed objects (primarily Microsoft's DCOM). As such, it is an ideal transport for method calls made over Web Services.

RPC-Oriented Web Services

Remote Procedure Calls (RPCs) made over Web-based protocols are essentially no different from those made over other protocols, such as IIOP, DCOM, or JRMP. The calls are usually synchronous (in other words, the client waits for the method to return before continuing). Zero or more parameters of varying types are passed into the call to provide information to process, and zero or more return values are generated to deliver the outputs of the remote method to the client. The remote method calls are delivered to some form of dispatcher at the remote server that determines which method should be called and arranges for the smooth flow of parameters and return values.

For RPC-style operation, SOAP implementations conform to the preceding description. The difference with SOAP (and other Web-based RPC mechanisms, such as XML-RPC) is that it uses standard, general-purpose transports, such as HTTP, together with a text-based method call description in XML. All of the parameters and return values are encoded in XML as part of the SOAP body, while information about the service and method to call are provided in the transport header and possibly the SOAP header. When sent over HTTP, the SOAP header and body are wrapped in another XML document—the SOAP envelope—and this envelope forms the body of an HTTP POST request.

An HTTP-based SOAP message will be delivered to a SOAP router that takes the form of an HTTP servlet (for a Java implementation). The SOAP router will examine the HTTP and SOAP header information and decide how it should forward the message body. This will involve instantiating or calling a particular component or class that will receive the message. The SOAP router, or its helper classes, will also perform the conversion of the XML-based parameters into Java objects and primitives that can be passed as part of the service invocation. Figure 20.5 shows the operation of such a SOAP router. Note that the description of the Web Service is used by both the client and server to help determine the correct mapping between Java and XML for method calls and parameter types.

This is all good, but why go to this effort? Why not use an existing RPC mechanism, such as RMI or just use HTTP itself?

The justification for not using RMI or CORBA relates to commonality and security. There are at least three different distributed object protocols (CORBA, RMI, and DCOM), each of which has its adherents. The use of HTTP and XML provides a common protocol that is not tied to any vendor. Also, the protocols listed have great difficulty in penetrating most firewalls (not surprising, given their ability to invoke random functionality). However, HTTP (and SMTP) have general right of access through most firewalls, which makes it easier to integrate applications across organizational boundaries (after the security questions are sorted out).

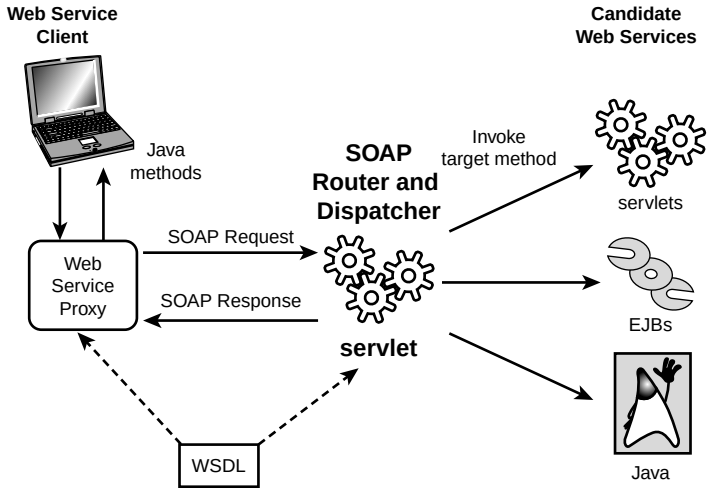


FIGURE 20.5
A Java-based SOAP router.

From a developer's perspective, one of SOAP's greatest assets is its ability to penetrate firewalls. However, from an administrator's point of view, this presents the same types of problem as traditional RPC, namely the ability to target a random function call at an exposed server. Although the principle of SOAP is only a small step on from the invocation of server-side functionality such as CGI, great care should be taken to ensure adequate security when exposing Web Services. The overall security story for Web Services is still a work in progress.

Although raw HTTP is a good transport, it was created to exchange simple HTML messages. This does not provide the sophistication required for a distributed invocation environment. The use of a defined XML message format brings structure to this environment and allows for the interoperability of Web Service clients and servers from different vendors—something that escaped CORBA until comparatively recently.

Now that you understand the architecture and motivation for RPC-style Web Services, you can install a Java-based Web Service environment and, through it, use and build your own Web Services.

Setting up Axis under Tomcat 4.0

The environment you will use for Web Service development in the first instance consists of the Tomcat servlet engine and the Axis Web Service toolkit, both from the Apache Software Foundation.

You can download Tomcat 4.0 from the Apache Software Foundation at `http://jakarta.apache.org/tomcat/` or install it from the CD-ROM as follows:

1. Unzip the Tomcat 4.0 archive (`jakarta-tomcat-4.0.1.zip`) into an appropriate directory on your hard drive (an example from Windows would be `C:\jakarta-tomcat-4.0.1`).
2. In your personal or system environment, set the environment variable `CATALINA_HOME` to point to this directory.

You can download Axis from the Apache Software Foundation or install it from the CD-ROM as follows:

- Unzip the Axis archive (`xml-axis-alpha2-bin.zip`) into an appropriate directory on your hard drive (an example from Windows would be `C:\axis-1_0`).
- Copy the `webapps\axis` directory from the `axis-1_0` distribution into Tomcat's `webapps` directory (`{CATALINA_HOME}\webapps`).

You need to install XML support for Axis from the Fall 01 JAX Pack (available from Sun or on the CD-ROM) as follows:

- Unzip the JAX Pack archive (`java_xml_pack-fall01.zip`) into an appropriate directory on your hard drive (an example from Windows would be `C:\java_xml_pack-fall01`).
- Copy `crimson.jar` and `xalan.jar` from the `jaxp-1.1.3` directory into `axis\WEB-INF\lib` under Tomcat's `webapps` directory (`{CATALINA_HOME}\webapps`).

Tomcat and Axis are now installed with the appropriate XML support.

In the next section, you will create a client for a simple `hello` Web Service. First, you must install and test this simple Web Service as follows:

1. Install the class required for the `HelloService` by copying the `webservices` directory from the CD-ROM directory `Day20\examples\HelloService` to `axis\WEB-INF\classes` under Tomcat's `webapps` directory (`{CATALINA_HOME}\webapps`).
2. Start Tomcat by running the `startup` script/batch file in the `{CATALINA_HOME}\bin` directory.
3. To ensure that Tomcat and Axis are installed correctly, start a Web browser and point it at `http://localhost:8080/axis/index.html`. You should see a welcome screen from Axis.

- Now deploy the hello server using the `deployit` batch file in the CD-ROM directory `Day20\examples\HelloService`.

Assuming that you had no errors, you have now deployed a simple Web Service called `MyHelloService`. In the long and distinguished tradition of curly-bracket-based languages, you will start with a variation on the `Hello World!` program.

Service Description Information

Your Web server now has a Web Service installed under it. The next step is to access that Web Service. However, before you can take advantage of the Web Service, you need the following information:

- *A definition of the service you are calling*—This information corresponds to the traditional interface definition for an RPC or RMI server. The interface definition contains information about the methods available, the number and types of parameters, the type of any return values, and definitions of any complex types used as parameters.
- *The location of the service*—This corresponds to the binding information used by RPC and RMI servers. This Web Service binding information lists the protocols over which you can call the available Web Service methods. For each supported protocol, there is also a URL indicating the location of a server that provides an implementation of that service for that protocol.

As you may have surmised by now, all of this information is provided by a WSDL description of the service.

Anatomy of a WSDL Document

The WSDL for `MyHelloService` is shown in Listing 20.1. It is worth taking a few moments to study this information because it provides a good insight into the way that Web Services work.

LISTING 20.1 WSDL for the Hello Service (`MyHelloService.wsdl`)

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2:
 3: <definitions
 4:   targetNamespace="http://localhost:8080/axis/services/MyHelloService"
 5:   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 6:   xmlns:serviceNS="http://localhost:8080/axis/services/MyHelloService"
 7:   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 8:   xmlns="http://schemas.xmlsoap.org/wsdl/">
 9:
10:   <message name="sayHelloToRequest">
11:     <part name="arg0" type="xsd:string"/>
12:   </message>
13:
14:   <message name="sayHelloToResponse">
```

LISTING 20.1 Continued

```
 15:     <part name="sayHelloToResult" type="xsd:string"/>
 16:   </message>
 17:
 18:   <portType name="HelloServerPortType">
 19:     <operation name="sayHelloTo">
 20:       <input message="serviceNS:sayHelloToRequest"/>
 21:       <output message="serviceNS:sayHelloToResponse"/>
 22:     </operation>
 23:   </portType>
 24:
 25:   <binding name="HelloServerSoapBinding"
➥       type="serviceNS:HelloServerPortType">
 26:     <soap:binding style="rpc"
➥             transport="http://schemas.xmlsoap.org/soap/http"/>
 27:     <operation name="sayHelloTo">
 28:       <soap:operation soapAction="" style="rpc"/>
 29:       <input>
 30:         <soap:body use="encoded"
 31:                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 32:                    namespace="MyHelloService"/>
 33:       </input>
 34:       <output>
 35:         <soap:body use="encoded"
 36:                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 37:                    namespace="MyHelloService"/>
 38:       </output>
 39:     </operation>
 40:   </binding>
 41:
 42:   <service name="HelloServer">
 43:     <port name="HelloServerPort"
➥      binding="serviceNS:HelloServerSoapBinding">
 44:       <soap:address
➥      location="http://localhost:8080/axis/services/MyHelloService"/>
 45:     </port>
 46:   </service>
 47:
 48: </definitions>
```

The document consists of the following sections:

- The XML prolog and root element (lines 1–8 and 48). The namespace declarations on the root element (`definitions`) show that all unqualified elements and attributes come from the WSDL schema. The `soap` prefix denotes types from the SOAP schema, while the `xsd` prefix denotes types from the W3C XML Schema definition. There is also a namespace defined for this service that is associated with the `serviceNS` prefix.

- WSDL `message` definitions (lines 10–16). These define two matched messages—a request and a response. The request (`sayHelloToRequest`) takes a single string parameter and the response (`sayHelloToResponse`) also returns a single string.
- WSDL `portType` definitions (lines 18–23). A `portType` is the equivalent of an interface definition. It contains one or more `operation` definitions, which in turn are built from the `message` definitions in the document. In this case, there is a single `operation` defined in the `HelloServerPortType` called `sayHelloTo`. This consists of the two messages, `sayHelloToRequest` and `sayHelloToResponse`, seen earlier.
- Now that you have an interface (`portType`), you can define the protocols over which that interface can be accessed. The binding element (lines 25–40) creates a binding, called `HelloServerSoapBinding`, between the `HelloServerPortType` and SOAP. Within this WSDL binding, a SOAP binding (`soap:binding`) is defined. Because SOAP can work with a variety of underlying transports and it can work in an RPC-centric or document-centric way, the attributes on the `soap:binding` indicate that it is an RPC-style binding that uses HTTP.

  The WSDL `operation` is then mapped to a SOAP `operation` with input and output `soap:body` elements defined to map the request and response.
- Finally, an instance of the service is defined in the WSDL `service` element (lines 42–46). A WSDL `service` contains a list of WSDL `port` elements. Each `port` element defines a specific instance of a server that conforms to one of the WSDL `binding`s defined earlier.

  Again, in the case of the simple Hello service, the `service` element (named `HelloServer`) contains a single WSDL port called `HelloServerPort`. This specifies that a server conforming to the `HelloServerSoapBinding` can be found at the given SOAP `address`, namely
  `http://localhost:8080/axis/service/MyHelloService`.

This is a very simple WSDL document defining a very simple service. WSDL documents are typically far longer and more complex. Because of this, WSDL is largely intended for manipulation by tools and applications.

Creating a Java Proxy from WSDL

Given the service description in Listing 20.1, the next step is to create a client that can use this service. The simplest way to do this is to have a tool generate a proxy for the service. This proxy will be a local object that will hide away a lot of the complexity associated with the mechanics of calling methods on the service.

You can apply the Apache Axis `Wsdl2java` tool to `MyHelloService.wsdl` as follows

```
java org.apache.axis.wsdl.Wsdl2java MyHelloService.wsdl
```

To run the tools and compile the files, you must have the JAR files from the `axis\WEB-INF\lib` directory on your classpath, namely `axis.jar`, `clutil.jar`, `crimson.jar`, `log4j-core.jar`, `wsdl4j.jar`, and `xalan.jar`.

This will generate three Java files:

- `HelloServerPortType.java` is a Java interface that represents the remote interface (or `portType` in WSDL terms). This is shown in Listing 20.2. Note that the interface looks like an RMI interface in that it extends `java.rmi.Remote`, and the method is defined as throwing `java.rmi.RemoteException`. The service proxy implements this interface, and the client should use the interface type to reference instances of the service proxy.
- `HelloServer.java` is a factory class that creates instances of the service proxy. This is shown in Listing 20.3. The client instantiates a factory and then calls the `getHelloServerPort` method to obtain a service proxy. Two forms of this method are provided—one that allows the client to specify the endpoint at which the service resides and the other that takes no arguments. The latter method will use the location information contained in the WSDL file when instantiating the service proxy.
- `HelloServerSoapBindingStub.java` is the service proxy itself. Note that by using a separate interface to represent the `portType` and a factory for the creation of the proxy, the same client code can be used, regardless of the particular protocol binding. The code for `HelloServerSoapBindingStub.java` is not shown here because it is very similar to the "raw" SOAP code you will see shortly.

LISTING 20.2 `HelloServerPortType.java`

```
 1: /**
 2:  * HelloServerPortType.java
 3:  *
 4:  * This file was auto-generated from WSDL
 5:  * by the Apache Axis Wsdl2java emitter.
 6:  */
 7:   7:
 8: public interface HelloServerPortType extends java.rmi.Remote {
 9:     public String sayHelloTo(String arg0) throws java.rmi.RemoteException;
10: }
```

LISTING 20.3 HelloServer.java

```
 1: /**
 2:  * HelloServer.java
 3:  *
 4:  * This file was auto-generated from WSDL
 5:  * by the Apache Axis Wsdl2java emitter.
 6:  */
 7:
 8: public class HelloServer {
 9:
10:     // Use to get a proxy class for HelloServerPort
11:     private final java.lang.String HelloServerPort_address =
12:                       "http://localhost:8080/axis/services/MyHelloService";
13:     public HelloServerPortType getHelloServerPort() {
14:        java.net.URL endpoint;
15:         try {
16:             endpoint = new java.net.URL(HelloServerPort_address);
17:         }
18:         catch (java.net.MalformedURLException e) {
19:             return null; // unlikely as URL was validated in wsdl2java
20:         }
21:         return getHelloServerPort(endpoint);
22:     }
23:
24:     public HelloServerPortType
➥                     getHelloServerPort(java.net.URL portAddress) {
25:         try {
26:             return new HelloServerSoapBindingStub(portAddress);
27:         }
28:         catch (org.apache.axis.SerializationException e) {
29:             return null; // ???
30:         }
31:     }
32: }
```

You can now write a client application that uses these classes. The code for such an application is shown in Listing 20.4. This application simply takes the name passed as a parameter and sends it to the `sayHelloTo` method of the Web Service. You can see the creation of the `HelloServer` service proxy factory on line 20. The client then calls the `getHelloServerPort` method to obtain an instance of the service proxy (line 23). The client can then call the `sayHelloTo` method passing the given parameter (line 28). This method invocation is wrapped in a `try-catch` block to catch any potential `RemoteException` that may occur.

LISTING 20.4 `HelloServerClient.java` Application That Uses Generated Service Proxy

```
 1: import java.rmi.RemoteException;
 2:
 3: public class HelloServerClient
 4: {
 5:   public static void main(String [] args)
 6:   {
 7:     String name = "unknown";
 8:
 9:     if (args.length != 1)
10:     {
11:         System.out.println("Usage: WebServiceSayHello <name>");
12:         System.exit(1);
13:     }
14:     else
15:     {
16:         name = args[0];
17:     }
18:
19:     // Instantiate the factory
20:     HelloServer factory = new HelloServer();
21:
22:     // Get a PortType that represents this particular service
23:     HelloServerPortType service = factory.getHelloServerPort();
24:
25:     try
26:     {
27:       // Call the service
28:       String response = service.sayHelloTo(name);
29:
30:       System.out.println(response);
31:     }
32:     catch(RemoteException ex)
33:     {
34:       System.out.println("Remote exception: " + ex);
35:     }
36:   }
37: }
```

To test out your client, you should:

1. Compile the client code.
2. Ensure that the service is running (both Tomcat and the Axis server).
3. Run the client (with the appropriate classpath settings) as shown

```
prompt> java HelloServerClient Fred
Hello Fred!
```

Calling the Web Service Through SOAP

You have now accessed the service through a service proxy based on WSDL. However, you can access the service directly through SOAP, should that be necessary. Indeed, some older toolkits may only provide a SOAP-level API and no WSDL-based tools, so this section looks quickly at how you would achieve the same effect directly with SOAP.

Listing 20.5 shows the code you would write under Apache SOAP 2.2 (the precursor to Axis) to call the Hello service using the SOAP API directly.

LISTING 20.5 `SoapSayHello.java` Using the Apache SOAP 2.2 API

```
 1: import java.net.*;
 2: import java.util.*;
 3: import org.apache.soap.*;
 4: import org.apache.soap.rpc.*;
 5:
 6: public class SoapSayHello
 7: {
 8:   private static String serviceUrn = "MyHelloService";
 9:   private static String soapRouterUrl =
➥                  "http://localhost:8080/axis/servlet/AxisServlet";
10:
11:   public static void main(String[] args)
12:   {
13:     String name = "unknown";
14:
15:     if (args.length != 1)
16:     {
17:         System.out.println("Usage: SoapSayHello <name>");
18:         System.exit(1);
19:     }
20:     else
21:     {
22:         name = args[0];
23:     }
24:
25:     URL url = null;
26:
27:     try
28:     {
29:       url = new URL(soapRouterUrl);
30:     }
31:     catch (MalformedURLException ex)
32:     {
33:       System.out.println("Exception: " + ex);
34:       System.exit(1);
35:     }
36:
```

LISTING 20.5 Continued

```
37:     Call call = new Call();
38:
39:     call.setTargetObjectURI(serviceUrn);
40:     call.setMethodName("sayHelloTo");
41:
42:     Vector params = new Vector();
43:
44:     params.addElement(new Parameter("name", String.class,
45:                                     name, Constants.NS_URI_SOAP_ENC));
46:     call.setParams(params);
47:
48:     Response response;
49:
50:     try
51:     {
52:       response = call.invoke(url, "");
53:     }
54:     catch (SOAPException e)
55:     {
56:       System.err.println("Caught SOAPException (" +
57:                          e.getFaultCode() + "): " +
58:                          e.getMessage());
59:       return;
60:     }
61:
62:     if (!response.generatedFault())
63:     {
64:       Parameter retVal = response.getReturnValue();
65:       Object value = retVal.getValue();
66:
67:       System.out.println(value != null ? "\n" + value : "I don't know.");
68:     }
69:     else
70:     {
71:       Fault fault = response.getFault();
72:
73:       System.err.println("Generated fault: ");
74:       System.out.println ("  Fault Code   = " + fault.getFaultCode());
75:       System.out.println ("  Fault String = " + fault.getFaultString());
76:     }
77:   }
78: }
```

The first thing to notice is that the endpoint URL is now split into the service name and the SOAP router (lines 8 and 9). This SOAP router URL must be turned into a `java.net.URL` (lines 25–35) for it to be used.

A SOAP `Call` is then instantiated (line 37) and populated with the service name (line 39) and the method name (line 40). The parameters for the call must be encoded as `Parameter` instances, specifying the parameter name, Java class, and encoding required. A `java.util.Vector` containing all of the parameters is then passed to the `Call` object (lines 42–46).

The call is then made to the SOAP server using the `invoke` method (line 52). This is where the SOAP router `URL` is passed in. A SOAP `Response` is returned from `invoke`.

Now the result must be deciphered (lines 62–76). This involves checking for an error, retrieving the `Parameter` object, extracting the actual returned object from it, and then casting this returned object to the appropriate type.

As you can see, the use of a proxy is preferable because it removes most of the complexity. This is why the Java APIs for creating and sending SOAP messages—JAX-RPC and JAXM—both work at a higher level than this. The benefits of using the WSDL-based proxy are that the client code is less complex, there is type safety by using the generated Java interface, and the client developer needs to know very little about the SOAP-level operations or indeed about SOAP itself.

A Half-Way House

There is a compromise that can be made between service-specific calling using a proxy and the use of raw SOAP. Axis provides a `ServiceClient` class that performs much of the code shown in Listing 20.5. In fact, all of the code from line 25 on can be effectively replaced by the following lines:

```
ServiceClient client = new ServiceClient(soapRouterUrl);

String response = (String)client.invoke(serviceUrn,
                                        "sayHelloTo",
                                        new Object [] { name });
```

The service address, method name, and the parameters are all passed into the `invoke` method. Note that the last argument in the code shown creates a new array of type `Object` and populates it with a single element, which is the `String` containing the name provided by the user.

In this case, there is a lot more flexibility than with the WSDL proxy, because the method name and parameter can be specified at runtime. This allows for dynamic interaction with discovered services. However, the code shown is preferable to the SOAP code in Listing 20.5 because the code surrounding the call setup has been largely simplified.

Dynamic calling will be examined further tomorrow in the discussion surrounding the use of directory services.

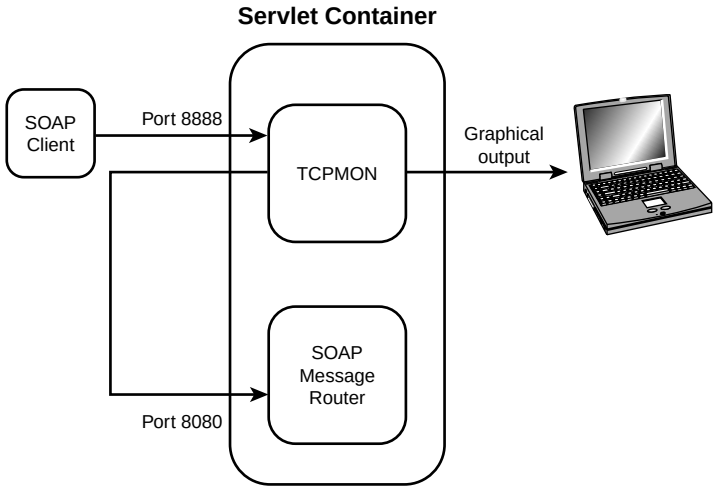
Debugging a SOAP Interaction

As with any distributed environment, debugging Web Service interaction is a challenge. One of the main issues is knowing precisely what is being sent and received. To assist with this, Axis provides a tool called `tcpmon` that will monitor and display SOAP traffic.

The basic idea is that you target your client at a different port. The `tcpmon` utility listens on that port, logs the SOAP traffic arriving, and then passes it on to the real SOAP server port. SOAP traffic sent back is also logged. If you cannot change the client configuration, you could change the port on which the SOAP server listens. The `tcpmon` utility can then listen on the original SOAP server port and forward traffic on to the new port. Figure 20.6 shows how an instance of `tcpmon` can monitor inbound traffic from SOAP clients on port 8888, log the traffic, and then forward it on to the real SOAP router listening on port 8080.

FIGURE 20.6

The `tcpmon` utility monitoring SOAP calls on port 8888 and passing them on to the real SOAP router on port 8080.



To start the `tcpmon` utility, type

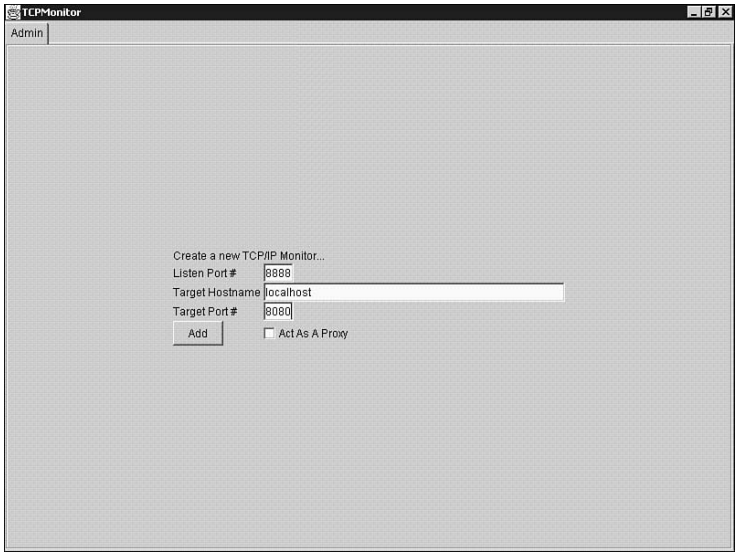
```
java org.apache.axis.utils.tcpmon
```

This will start a GUI through which the traffic will be displayed. When the GUI starts up, you will be prompted for the port on which to listen and also the port and host to which traffic should be forwarded. Figure 20.7 shows a monitor session being started that will listen on port 8888 and forward all traffic received on to `localhost:8080`.

The `tcpmon` utility allows you to set up multiple port/host/port mappings. Each will be displayed in its own tabbed pane.

FIGURE 20.7

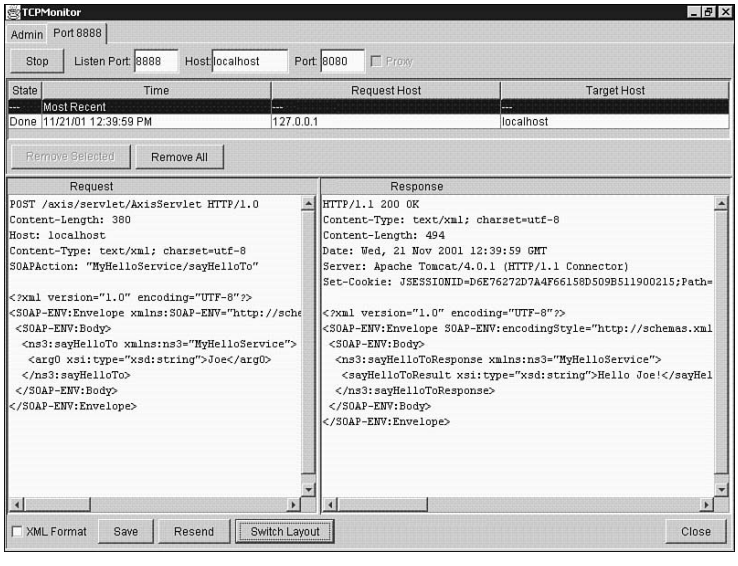
Setting up the port configuration for `tcpmon`.



The request and response messages are displayed as pairs, as shown in Figure 20.8. This shows an interaction between the Hello service and a client that has been modified so that its target port is configurable. The client sends its request to port 8888 with a `SOAPAction` of `MyHelloService/sayHelloTo`. You can see the method invocation and the string parameter in the SOAP body. The response is shown in the right pane. In the response, the SOAP body contains a `sayHelloToResponse` message encapsulating the `sayHelloToResult` return value.

FIGURE 20.8

SOAP request and response to the Hello service seen through `tcpmon`.



The `tcpmon` utility will retain a history of messages sent back and forth through a particular port. You can then look back through a sequence of messages in your own time.

Implementing an RPC-Style SOAP-Based Web Service

Now that you are familiar with writing a simple client for a Web Service, you will probably want to create your own Web Service in Java.

To deliver a Web Service, you must provide the following:

- The business logic
- A description of the Web Service, such as its name, the methods to be exposed and so forth
- A router to receive SOAP calls and dispatch method calls to the business logic

Following the same principles as EJBs, it would be good if most of the Web Service-related functionality was provided for you, leaving you to concentrate on the business logic. Ideally, you would provide the business logic and some of the Web Service description, leaving someone else to provide the rest. Fortunately, as you will see, the Axis environment provides most of the Web Service plumbing, as do other Java-based Web Service containers.

Wrapping up a Java class as a Web Service

To create a Web Service to run under Axis, all you need to do is supply a Java class and some configuration information. The Java class needs no specific code to make itself Web Service-aware, simply one or more public methods.

As an example, consider the `SimpleOrderServer` shown in Listing 20.6. This contains the business logic to be wrapped. Note that in this instance, the server is just a standard Java class. Apart from the package name, there is no indication that this is intended to be a Web Service. Even the package name is there only to separate this example from other classes. This class will be instantiated and its methods invoked by the Axis server.

The Axis server takes the form of a servlet called, not surprisingly, `AxisServlet`. This servlet acts as the router for all HTTP-based SOAP requests and also supplies WSDL descriptions for deployed services, as you will see later. The `AxisServlet` can be found at `http://localhost:8080/axis/servlet/AxisServlet`.

LISTING 20.6 `SimpleOrderServer.java`—A Simple Server to Receive and Process Orders

```
 1: package webservices;
 2:
 3: public class SimpleOrderServer
 4: {
 5:   public String submitOrder(String customerID, String productCode,
➥                                                          int quantity)
 6:   {
 7:     // Form up a receipt for the order
 8:     String receipt = "";
 9:
10:     receipt = "Thank you, " + customerID + "\n";
11:     receipt += "You ordered " + quantity + " " + productCode + "'s\n";
12:     receipt += "That will cost you " + (quantity * 50) + " Euros";
13:
14:     return receipt;
15:   }
16: }
```

Now that you have your business logic, you will need to provide some information for the `AxisServlet`:

- The name under which the service is to be deployed—in this case, the name will be `SimpleOrderService`.
- The class that provides the functionality for the service—in this case, this is the `SimpleOrderServer` as shown in Listing 20.6.
- The names of the methods that should be exposed as part of the service—in this case, the single method `submitOrder`.

This information is encapsulated in XML format, as shown in Listing 20.7. The `<service>` element defines the name and the fact that this is an RPC-based service. The `<option>` elements define the class and method names.

LISTING 20.7 Deployment Descriptor for the `SimpleOrderService` (`deploy_simple_order.xml`)

```
1: <admin:deploy xmlns:admin="AdminService">
2:  <service name="SimpleOrderService" pivot="RPCDispatcher">
3:   <option name="className" value="webservices.SimpleOrderServer"/>
4:   <option name="methodName" value="submitOrder"/>
5:  </service>
6: </admin:deploy>
```

The deployment descriptor syntax shown works with Axis alpha 2. However, there is a stated commitment that this syntax will migrate to a standard Web Service Deployment Descriptor (WSDD) syntax at a later date. Although the syntax will differ, the principles will remain largely the same.

You are now ready to deploy your Web Service.

First, copy over the `SimpleOrderServer.class` file to the Axis classes directory:

```
{TOMCAT_HOME}\webapps\axis\WEB-INF\classes\
```

This ensures that the class is on the Axis classpath so that the server can find it when you invoke the `submitOrder` method. Make sure that you retain the appropriate directory hierarchy. This means that because `SimpleOrderServer` is in the `webservices` package, it should appear as `webservices\SimpleOrderServer.class` below the Axis classes directory.

Next, use the `ServiceManagerClient` to deploy your Web Service according to the deployment descriptor in Listing 20.7, as follows:

```
java org.apache.axis.client.AdminClient
➥ -lhttp://localhost:8080/axis/servlet/AxisServlet deploy_simple_order.xml
```

Assuming that you get no errors, you can list the services currently deployed, either by pointing your Web browser at the Axis services URL, `http://localhost:8080/axis/services?list`, or by issuing the following command from the command line:

```
java org.apache.axis.client.AdminClient
➥ -lhttp://localhost:8080/axis/services list
```

The URL prefix `/axis/services` is simply a mapping for the `/axis/servlet/AxisServlet` URL prefix, so they can be used interchangeably.

However, be aware that the `web.xml` mapping for the virtual directory `services` is incorrect on Axis alpha 2. To list a service or obtain its WSDL, you must either update the `web.xml` file for the `services` mapping so that it looks as follows:

```
<servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

or always use the explicit `AxisServlet` URL:

```
http://localhost:8080/axis/servlet/AxisServlet/{service name and
options}
```

Whether you list the services through the Web browser or the tool, you should see a list of services similar to the one in Listing 20.8. You can see the deployment information for the `SimpleOrderService` between lines 24–27, and that for `MyHelloService` that you used earlier between lines 29–32.

LISTING 20.8 List of Services Deployed under Axis

```
 1: <engineConfig>
 2:   <handlers>
 3:     <handler class="org.apache.axis.handlers.http.HTTPAuthHandler"
➥                                                    name="HTTPAuth"/>
 4:     <handler class="org.apache.axis.handlers.DebugHandler" name="debug"/>
 5:     <handler class="org.apache.axis.handlers.EchoHandler"
➥                                                  name="EchoHandler"/>
 6:     <handler class="org.apache.axis.handlers.JWSProcessor"
➥                                                  name="JWSProcessor"/>
 7:     <handler class="org.apache.axis.providers.java.RPCProvider"
➥                                                 name="RPCDispatcher"/>
 8:     <chain flow="JWSHandler,debug" name="global.request"/>
 9:     <chain flow="Authenticate,Authorize" name="authChecks"/>
 10:     <handler class="org.apache.axis.handlers.http.URLMapper"
➥                                                    name="URLMapper"/>
 11:     <handler class="org.apache.axis.handlers.SimpleAuthorizationHandler"
➥                                                    name="Authorize"/>
 12:     <handler class="org.apache.axis.handlers.JWSHandler"
➥                                                   name="JWSHandler"/>
 13:     <handler class="org.apache.axis.providers.java.MsgProvider"
➥                                                 name="MsgDispatcher"/>
 14:     <handler class="org.apache.axis.transport.local.LocalResponder"
➥                                                name="LocalResponder"/>
 15:     <handler class="org.apache.axis.handlers.SimpleAuthenticationHandler"
➥                                                  name="Authenticate"/>
 16:   </handlers>
 17:
 18:   <services>
 19:     <service pivot="MsgDispatcher" name="AdminService">
 20:       <option name="methodName" value="AdminService"/>
 21:       <option name="enableRemoteAdmin" value="false"/>
 22:       <option name="className" value="org.apache.axis.utils.Admin"/>
 23:     </service>
 24:     <service pivot="RPCDispatcher" name="SimpleOrderService">
 25:       <option name="methodName" value="submitOrder"/>
 26:       <option name="className" value="webservices.SimpleOrderServer"/>
 27:     </service>
 28:     <service pivot="JWSProcessor" name="JWSProcessor"/>
 29:     <service pivot="RPCDispatcher" name="MyHelloService">
 30:       <option name="methodName" value="sayHelloTo"/>
 31:       <option name="className" value="webservices.HelloServer"/>
 32:     </service>
```

LISTING 20.8 Continued

```
33:     <service pivot="EchoHandler" name="EchoService"/>
34:   </services>
35:   <transports>
36:     <transport request="URLMapper" name="SimpleHttp"/>
37:     <transport request="HTTPAuth,URLMapper" name="http"/>
38:     <transport response="LocalResponder" name="local"/>
39:   </transports>
40: </engineConfig>
```

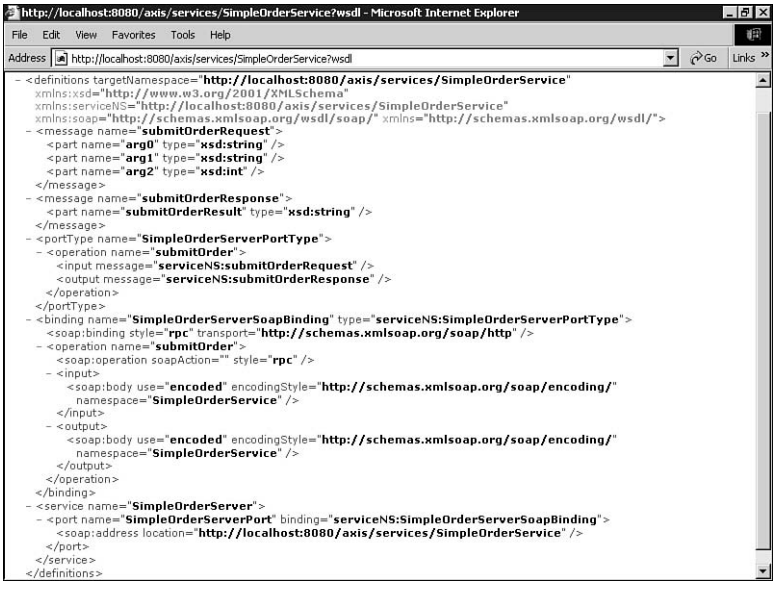
A Client for Your Web Service

Now that you have successfully deployed your server, you can create a client for it as you did for the `MyHelloService` earlier. Again, you have the choice of directly using a `ServiceClient` or generating a client-side service proxy based on the service's WSDL.

But wait, it is all very well to talk about generating a service proxy from WSDL, but the service you have just deployed is just a Java class. How do you get the WSDL for it? Well, once again the tools can help here. The Axis environment will provide for you the WSDL description if you append the query string `"?wsdl"` onto the service URL. Figure 20.9 shows the WSDL for the `SimpleOrderService` displayed in a Web browser. You can then simply save the page onto local disk as `SimpleOrderService.wsdl`.

FIGURE 20.9

Obtaining the WSDL description of your service in a Web browser through Axis.



Other tools provide other ways of obtaining the WSDL from a Java-based service. For example, IBM's WSTK provides a graphical tool called `wsdlgen` that allows you to select the Java methods that you want to expose as Web Service methods.

After you have a WSDL file for your service, you can create a client as you did before for the `MyHelloService` by using `Wsdl2java` to generate client-side service proxy classes (`SimpleOrderServer`, `SimpleOrderServerPortType`, and `SimpleOrderServerSoapBindingStub`, in this case).

It is a good idea to generate your service proxy classes in a different directory from your server. Unless you are very careful with your naming strategy, `Wsdl2java` can easily select the same name for its service proxy factory as you have for your server (`SimpleOrderServer.java`). Creating the client in a different directory can save a lot of frustration.

The client code to use the `SimpleOrderServer` client-side service proxy is shown in Listing 20.9.

LISTING 20.9 `SimpleOrderClient.java`—Takes Parameters for the Customer ID, Product Code, and Quantity and Submits the Order to the `SimpleOrderService`

```
 1: import java.rmi.RemoteException;
 2:
 3: public class SimpleOrderClient
 4: {
 5:   public static void main(String [] args)
 6:   {
 7:     String customerId = "unknown";
 8:     String productCode = "Widget";
 9:     int quantity = 1;
10:
11:     if (args.length != 3)
12:     {
13:          System.out.println("Usage: SimpleOrderClient <customerId>
➥                                          <productCode> <quantity>");
14:         System.exit(1);
15:     }
16:     else
17:     {
18:         customerId = args[0];
19:         productCode = args[1];
20:         quantity = Integer.parseInt(args[2]);
21:     }
22:
23:     // Intantiate the factory
```

LISTING 20.9 Continued

```
24:     SimpleOrderServer factory = new SimpleOrderServer();
25:
26:     // Get a PortType that represents this particular service
27:     SimpleOrderServerPortType service = factory.getSimpleOrderServerPort();
28:
29:     try
30:     {
31:       // Call the service
32:       String response = service.submitOrder(customerId, productCode,
➥                                                          quantity);
33:
34:       System.out.println(response);
35:     }
36:     catch(RemoteException ex)
37:     {
38:       System.out.println("Remote exception: " + ex);
39:     }
40:   }
41: }
```

The following shows how you would run the client and the result returned:

```
prompt> java SimpleOrderClient Acme ACX-09387 37
Thank you, Acme
You ordered 37 ACX-09387's
That will cost you 1850 Euros
```

You have now created a complete Java Web Service client and server.

Starting from WSDL

When a system is designed, the designers will create UML diagrams (or the like) to represent the system entities and interactions between them. Tools can then generate programming artefacts, such as Java classes and interfaces based on this information. If the system will be based on Web Services, such artefacts will include WSDL descriptions of the required services. You, as a Java developer, may then be presented with a WSDL description that requires a Java implementation.

Rather than having to work out manually what sort of Java class would match that WSDL description, the Java-based Web Service toolkits provide utilities that can produce the appropriate Java skeleton code from a given WSDL document. Under Axis, the `Wsdl2java` utility can be used to produce a skeleton Java Web Service as follows:

```
java org.apache.axis.wsdl.Wsdl2java --skeleton SimpleOrderServer.wsdl
```

This generates all of the client-side service proxy files together with the following server-side files:

- `SimpleOrderServerSoapBindingSkeleton.java` This is the class that is deployed as the target for the `AxisServlet`. You will not edit this file, but its contents are shown in Listing 20.10. As you can see, it delegates the business functionality to an instance of the `SimpleOrderServerSoapBindingImpl` class that is creates (lines 14 and 25).
- `SimpleOrderServerSoapBindingImpl.java` The implementation class is the one you will fill with the business logic. The skeleton provided is shown in Listing 20.11. You can see the location for the business logic on line 14.
- `deploy.xml` and `undeploy.xml` Two XML deployment files are provided—one to deploy the service and one to undeploy it. In Listing 20.12, you can see that the service is deployed as `SimpleOrderServerPort` and calls the `SimpleOrderServerSoapBindingSkeleton` class to service requests. You can use this XML file together with `AdminClient` to deploy the Web Service.

LISTING 20.10 `SimpleOrderServerSoapBindingSkeleton.java`—A Server-Side Proxy for the Given WSDL Service Description

```
 1: /**
 2:  * SimpleOrderServerSoapBindingSkeleton.java
 3:  *
 4:  * This file was auto-generated from WSDL
 5:  * by the Apache Axis Wsdl2java emitter.
 6:  */
 7:
 8: public class SimpleOrderServerSoapBindingSkeleton
 9: {
10:     private SimpleOrderServerPortType impl;
11:
12:     public SimpleOrderServerSoapBindingSkeleton()
13:     {
14:         this.impl = new SimpleOrderServerSoapBindingImpl();
15:     }
16:
17:     public
➥ SimpleOrderServerSoapBindingSkeleton(SimpleOrderServerPortType impl)
18:     {
19:         this.impl = impl;
20:     }
21:
22:     public Object submitOrder(String arg0, String arg1, int arg2)
23:                                             throws java.rmi.RemoteException
24:     {
```

LISTING 20.10 Continued

```
25:         Object ret = impl.submitOrder(arg0, arg1, arg2);
26:         return ret;
27:     }
28: }
```

LISTING 20.11 `SimpleOrderServerSoapBindingImpl.java`—A Skeleton Within Which to Implement Your Business Logic

```
 1: /**
 2:  * SimpleOrderServerSoapBindingImpl.java
 3:  *
 4:  * This file was auto-generated from WSDL
 5:  * by the Apache Axis Wsdl2java emitter.
 6:  */
 7:
 8: public class SimpleOrderServerSoapBindingImpl
 9:                                 implements SimpleOrderServerPortType
10: {
11:     public String submitOrder(String arg0, String arg1, int arg2)
12:                                     throws java.rmi.RemoteException
13:     {
14:         throw new java.rmi.RemoteException ("Not Yet Implemented");
15:     }
16: }
```

LISTING 20.12 `deploy.xml`—A Deployment Descriptor Automatically Generated from the `SimpleOrderServer` WSDL

```
 1: <!--                                                                -->
 2: <!--Use this file to deploy some handlers/chains and services  -->
 3: <!--Two ways to do this:                                            -->
 4: <!--  java org.apache.axis.utils.Admin deploy.xml                 -->
 5: <!--     from the same dir that the Axis engine runs                -->
 6: <!--or                                                              -->
 7: <!--  java org.apache.axis.client.AdminClient deploy.xml          -->
 8: <!--     after the axis server is running                           -->
 9: <!--This file will be replaced by WSDD once it's ready              -->
10:
11: <m:deploy xmlns:m="AdminService">
12:
13:    <!-- Services from SimpleOrderServer WSDL service -->
14:
15:    <service name="SimpleOrderServerPort" pivot="RPCDispatcher">
16:       <option name="className"
➥                        value="SimpleOrderServerSoapBindingSkeleton"/>
```

LISTING 20.12 Continued

```
17:       <option name="methodName" value=" submitOrder"/>
18:    </service>
19: </m:deploy>
```

Using Axis JWS files

As you have seen, most of the work on the server side is done for you by the tools and utilities that come with the toolkit. All you have to really provide is a Java class and everything else can be generated for you. Given this, the Axis project has taken things one stage further and developed the concept of JWS (or `.jws`) files.

JWS files (short for Java Web Service) provide a way of deploying a Java-based Web Service in the simplest possible way. All you do is take the Java class that would be the target for the Web Service deployment descriptor (the `class` option under the `<service>` element) and change its suffix to `.jws` instead of `.java`. You then place this file somewhere below the `/axis` directory and that's it. There is no need to create deployment information or to use `AdminClient`.

One thing to note is that if your classes belong to a package hierarchy, you will have to place them in the appropriate subdirectory under the axis directory.

As an example, consider creating a JWS service based on the `SimpleOrderServer`. Take the `SimpleOrderServer.java` file and rename it as `SimpleOrderServer2.java` (and rename the class inside it to `SimpleOrderServer2` also, as shown in Listing 20.13). The change to the filename and classname will avoid any confusion with the original service. Then change the file extension from `.java` to `.jws` and copy the file into the `/axis` directory. You can then access it as follows:

```
http://localhost:8080/axis/SimpleOrderServer2.jws
```

LISTING 20.13 `SimpleOrderServer2.jws`—A JWS Version of the Simple Order Server

```
 1: public class SimpleOrderServer2
 2: {
 3:   public String submitOrder(String customerID, String productCode,
➥                                                          int quantity)
 4:   {
 5:     // Form up a receipt for the order
 6:     String receipt = "";
 7:
 8:     receipt = "Thank you, " + customerID + "\n";
 9:     receipt += "You ordered " + quantity + " " + productCode + "'s\n";
10:     receipt += "That will cost you " + (quantity * 50) + " Euros";
11:
```

LISTING 20.13 Continued

```
12:     return receipt;
13:   }
14: }
```

A JWS file is a fully-functioning Web Service, so you can obtain its WSDL by using the URL `http://localhost:8080/axis/SimpleOrderServer2.jws?wsdl`.

The result of this is shown in Listing 20.14. As you can see, the names used reflect the name of the JWS file, such as `SimpleOrderServer2` as the name of the service on line 38. The SOAP address provided as part of the `port` targets the JWS file, as you can see on line 41. You can then use this WSDL information to create a client-side service proxy and call the service as you have done before.

LISTING 20.14 WSDL Generated from `SimpleOrderServer2.jws`

```
 1: <?xml version="1.0" encoding="UTF-8" ?>
 2:   <definitions
➥   targetNamespace="http://localhost:8080/axis/SimpleOrderServer2.jws"
 3:       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 4:       xmlns:serviceNS="http://localhost:8080/axis/SimpleOrderServer2.jws"
 5:       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 6:       xmlns="http://schemas.xmlsoap.org/wsdl/">
 7:     <message name="submitOrderResponse">
 8:       <part name="submitOrderResult" type="xsd:string" />
 9:     </message>
10:     <message name="submitOrderRequest">
11:       <part name="arg0" type="xsd:string" />
12:       <part name="arg1" type="xsd:string" />
13:       <part name="arg2" type="xsd:int" />
14:     </message>
15:   <portType name="SimpleOrderServer2PortType">
16:     <operation name="submitOrder">
17:       <input message="serviceNS:submitOrderRequest" />
18:       <output message="serviceNS:submitOrderResponse" />
19:       </operation>
20:   </portType>
21:   <binding name="SimpleOrderServer2SoapBinding"
22:            type="serviceNS:SimpleOrderServer2PortType">
23:     <soap:binding style="rpc"
➥              transport="http://schemas.xmlsoap.org/soap/http" />
24:     <operation name="submitOrder">
25:       <soap:operation soapAction="" style="rpc" />
26:       <input>
27:         <soap:body use="encoded"
28:                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
29:                    namespace="" />
30:       </input>
```

LISTING 20.14 Continued

```
31:       <output>
32:         <soap:body use="encoded"
33:                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
34:                  namespace="" />
35:       </output>
36:     </operation>
37:   </binding>
38:   <service name="SimpleOrderServer2">
39:     <port name="SimpleOrderServer2Port"
40:           binding="serviceNS:SimpleOrderServer2SoapBinding">
41:       <soap:address
➥        location="http://localhost:8080/axis/SimpleOrderServer2.jws" />
42:     </port>
43:   </service>
44: </definitions>
```

JWS files are a very neat idea, and they remove a lot of the hassle from creating and deploying Web Services. However, there is a cost to the simplicity compared to the Web Services seen previously, including

- All public methods on the JWS class are exposed; you have no fine-grained control as you would with a deployment descriptor.
- You cannot control the lifetime of the JWS class, so you cannot maintain session state between invocations.

This means that "serious" Web Services are likely to be deployed as Java classes (or wrapped EJBs). However, in the same way that JSPs provide a lightweight way of using a servlet, JWS files provide a lightweight way of delivering Web Services. Other factors, such as the ease of deployment and automatic compilation reinforce the similarities between these two models.

Session Context and Web Services

As with servlets and JSPs, it is important to be able to maintain session state between method invocations on a particular Web Service. To do this, Axis lets you control state management policy at both the client and server side.

If a server is intended to allow state to be maintained between method invocations, this needs to be specified in the deployment descriptor. Listing 20.15 shows a variant of the `SimpleOrderService` that maintains state between method invocations. This class stores the name of the last customer to call the `submitOrder` method (line 23) and checks the name of the next customer against this value (line 12). If the customer name is the same, the message in the receipt is different (lines 14–18). Obviously, if there is no session maintained, the instance of the server class will be discarded after each method call.

The only time that the "Hello again" greeting will be seen is if session state has been maintained between invocations.

LISTING 20.15 `SessionSimpleOrderServer.java`—A Version of the Simple Order Server That Maintains Session State

```
 1: package webservices;
 2:
 3: public class SessionSimpleOrderServer
 4: {
 5:   private String lastCustomer = "";
 6:
 7:   public
➥String submitOrder(String customerID, String productCode, int quantity)
 8:   {
 9:     // Form up a receipt for the order
10:     String receipt = "";
11:
12:     if (customerID.equals(lastCustomer))
13:     {
14:       receipt = "Hello again, " + customerID + "\n";
15:     }
16:     else
17:     {
18:       receipt = "Thank you, " + customerID + "\n";
19:     }
20:     receipt += "You ordered " + quantity + " " + productCode + "'s\n";
21:     receipt += "That will cost you " + (quantity * 50) + " Euros";
22:
23:     lastCustomer = customerID;
24:
25:     return receipt;
26:   }
27: }
```

To provide the ability to maintain session state between invocations, an option must be set in the deployment descriptor to indicate that the Web Service implementation should maintain session state. The deployment descriptor for the `SessionSimpleOrderServer` is shown in Listing 20.16, and you can see on line 6 that a new `option` element has been added to set the `scope` option to `Session`. This indicates to the Axis server that the server instance should not be discarded until the user session has ended.

LISTING 20.16 Deployment Descriptor for the `SessionSimpleOrderServer`

```
1: <admin:deploy xmlns:admin="AdminService">
2:   <service name="SessionSimpleOrderService" pivot="RPCDispatcher">
```

LISTING 20.16 Continued

```
  3:     <option name="className"
➥          value="webservices.SessionSimpleOrderServer" />
  4:     <option name="methodName" value="submitOrder" />
  5:     <option name="scope" value="Session"/>
  6:   </service>
  7: </admin:deploy>
```

Under Axis, simply setting state on the server side will have no effect unless the client also indicates that it wants to maintain session state. To do this, the client calls the `setMaintainSession` on the client-side service proxy, as shown in the altered client in Listing 20.17 on lines 30–32.

LISTING 20.17 Client Code Altered to Work with the `SessionSimpleOrderServer` to Maintain Session State

```
 1: import java.rmi.RemoteException;
 2:
 3: public class SessionSimpleOrderClient
 4: {
 5:   public static void main(String [] args)
 6:   {
 7:     String customerId = "unknown";
 8:     String productCode = "Widget";
 9:     int quantity = 1;
10:
11:     if (args.length != 3)
12:     {
13:         System.out.println("Usage: SessionSimpleOrderClient " +
14:       "<customerId> <productCode> <quantity>");
15:         System.exit(1);
16:     }
17:     else
18:     {
19:         customerId = args[0];
20:         productCode = args[1];
21:         quantity = Integer.parseInt(args[2]);
22:     }
23:
24:     // Intantiate the factory
25:     SessionSimpleOrderServer factory = new SessionSimpleOrderServer();
26:
27:     // Get a PortType that represents this particular service
28:     SessionSimpleOrderServerPortType service =
➥                          factory.getSessionSimpleOrderServerPort();
29:
30:     SessionSimpleOrderServerSoapBindingStub stub =
31:                          (SessionSimpleOrderServerSoapBindingStub)service;
```

LISTING 20.17 Continued

```
32:     stub.setMaintainSession(true);
33:
34:     try
35:     {
36:       // Call the service
37:       String response;
38:
39:       response = service.submitOrder(customerId, productCode, quantity);
40:
41:       System.out.println(response);
42:
43:       response = service.submitOrder(customerId, productCode, quantity);
44:
45:       System.out.println(response);
46:     }
47:     catch(RemoteException ex)
48:     {
49:       System.out.println("Remote exception: " + ex);
50:     }
51:   }
52: }
```

When the two consecutive calls to `submitOrder` are made (as seen on lines 39–45), the server instance is not discarded between the calls, so the following output is seen:

```
prompt> java SessionSimpleOrderClient Fred ASX4220 15
Thank you, Fred
You ordered 15 ASX4220's
That will cost you 750 Euros
Hello again, Fred
You ordered 15 ASX4220's
That will cost you 750 Euros
```

Maintaining state is particularly useful when building up information from a client or when expensive resources, such as EJB references, are required.

In this context, the term "expensive" is used to denote that a large amount of valuable connection time is wasted obtaining such a resource if it has to be created or retrieved for every client call. Ideally, the server-side implementation should be able to cache and recycle such resources between clients to make best use of them. Failing that, you would certainly want to retain resources between client invocations that form part of the same workflow (or transaction).

Issues surrounding resource recycling and scalability have already been discussed on Day 18, "Patterns."

Wrapping Existing J2EE Functionality as Web Services

As noted earlier, many Web Service implementations will be used to wrap existing functionality. If that functionality takes the form of JavaBeans (such as those used in combination with servlets or JSPs), it is relatively simple to use these from the Web Service Java classes seen so far. In this case, the Java class provided as the target for the Web Service router will perform a similar role to the servlet or JSP in that it provides the front-end interaction logic that will then draw on functionality in the JavaBeans as required. This Java class is then acting in the role of a façade, as discussed on Day 18.

The key question for J2EE applications is how Web Services will interact with the J2EE container and server to provide access to J2EE resources, most notably EJBs. The answer is relatively simple, although there are some complications in the short term.

As you have seen, the Web Service router takes the form of a servlet, such as the `AxisServlet`. Hence, your Web Service Java classes are the equivalent of JavaBeans or helper classes being used by any other servlet. This means that they are being invoked within the context of the servlet. Because the Web Service router can run within a J2EE servlet container, it can be installed on a J2EE server. This means that the router and any classes used by it can gain access to J2EE resources through the container, so your Web Service Java classes can use the same code as your servlets and JSPs to access EJBs and other J2EE resources.

All of this seems quite simple, but there are some issues with this model:

- To deploy your Web Services under J2EE, they must be bundled in a `WAR` file. This Web Application will contain mappings for the endpoint addresses used by your client applications, such as `http://acme.com:8080/axis/services/MyService`. The virtual directory mappings used here (`/axis` and `/axis/services`) can only be used by one Web Application in the server. This means that any Web Services that use these endpoint addresses must be deployed as part of the same Web Application. Also, to comply with the J2EE model, all J2EE resources accessed from within this Web Application must be declared as part of the deployment descriptor.

  The end result of this is that your Web Service Java classes must be bundled into the `WAR` alongside the Axis classes. Any change to any of your Web Services, or the addition of any new Web Services, will require this `WAR` to be re-built and re-deployed. This is slightly inconvenient because it may cause the re-deployment of completely unrelated services that are running quite happily. The only alternative to this involves deploying multiple copies of the Web Service router, each with different endpoint mappings. Over time, as Web Services become incorporated into the underlying J2EE platform, this issue should disappear.

- To avoid the previous issue and provide a single Web Service router that is independent of the Web Services themselves, you could use an external servlet container, such as Tomcat, to house your Web Service router. This would allow you to deploy Web Services simply by copying their classes into place and informing the router of their existence. In this case, you would need to access EJBs and other J2EE resources from this remote container. In some respects, this is not a problem because you can initialize your JNDI runtime to perform its resource lookups using the naming service on the remote J2EE server. The following code shows how you could look up an EJB deployed on the R2EE RI from Tomcat:

```
Hashtable env = new Hashtable();

env.put("java.naming.factory.initial",
➥"com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url", "iiop://localhost:1050");

InitialContext ic = new InitialContext(env);
Object lookup = ic.lookup(agencyJNDI);

home = (AgencyHome)PortableRemoteObject.narrow(lookup, AgencyHome.class);
```

  However, there is a problem with this in that the calling container must be able to propagate the appropriate security and transaction context to the remote EJB container. Because such propagation can still be the source of interoperability issues between J2EE servers, the configuration of an out-of-server EJB client can become tricky. As a result, this is not an ideal solution at present.

- The previous two issues are largely technical in nature and can generally be overcome by the application of suitable mechanisms. However, there is a more central issue here. Regardless of where the Web Services are deployed or invoked, you still have to write a Web Service Java class to expose the business logic in your EJBs.

  In some cases, this may be desirable in that you want to apply extra constraints on Web Service-based users of these EJBs. However, if the EJB is a Session EJB containing the business logic you want to expose, you would want a way of automatically creating the intervening Web Service Java class. After all, the EJB interface is just a list of Java methods similar to those on the Web Service Java class.

  The ability to expose EJB methods automatically as Web Services is provided in Axis and its predecessor, Apache SOAP. The “pluggable provider” mechanism allows you to use an EJB provider rather than a Java class-based one. You then specify this in the deployment descriptor, and the SOAP engine will then target the given EJB when a SOAP call arrives. Similarly, the IBM WSTK provides a graphical way of wrapping up EJBs as Web Services, although as of October 2001, this would only work with EJB-JAR files conforming to the EJB 1.0 specification. In both cases, there is still the issue of ensuring the correct context propagation to the target J2EE server.

As you can see, these issues make the creation and deployment of J2EE-based Web Services slightly awkward at the moment. However, the primary purpose of J2EE 1.4 is the incorporation of Web Services into the core J2EE platform. This should mean that creating and deploying a Web Service that uses J2EE resources becomes as easy as creating a servlet or EJB.

Parameter Types and Type Mapping

So far, the Web Services you have seen have used simple parameters, such as strings and integers. However, in the real world, most systems will need to pass complex types such as arrays, classes, and data structures. During the rest of today, you will look at how such complex types can be mapped between Java and SOAP/WSDL and create an order service that uses these types.

Mapping Between Java and SOAP/WSDL Types

For simple types, SOAP and WSDL use the representations defined in "XML Schema Part 2: Datatypes" that is part of the W3C XML Schema standard. There is a straight mapping for all Java primitive types except for `char`. There is also a straight mapping for the Java `String` class.

If you were to define a Web Service with the following (unlikely) method:

```
public void test(byte byteArg, short shortArg, int intArg, long longArg,
                 float floatArg, double doubleArg, char charArg,
                 boolean boolArg, String stringArg)
```

this would map into WSDL as follows:

```
<definitions targetNamespace="http://localhost:8080/axis/Test.jws"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:serviceNS="http://localhost:8080/axis/Test.jws"
             xmlns:ns1="java"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="testResponse">
    <part name="testResult" type="ns1:void"/>
  </message>
  <message name="testRequest">
    <part name="arg0" type="xsd:byte"/>
    <part name="arg1" type="xsd:short"/>
    <part name="arg2" type="xsd:int"/>
    <part name="arg3" type="xsd:long"/>
    <part name="arg4" type="xsd:float"/>
    <part name="arg5" type="xsd:double"/>
    <part name="arg6" type="ns1:char"/>
    <part name="arg7" type="xsd:boolean"/>
    <part name="arg8" type="xsd:string"/>
  </message>
```

Notice that all of the arguments except `char` are mapped to a type using the `xsd` prefix, which refers to the `http://www.w3.org/2001/XMLSchema` namespace.

However, when you start to work with arrays and complex Java types, such as your own classes, more effort must be put into the representation of these mappings. Consider what is done by RMI when you pass Java objects between a client and server:

- RMI uses the Java serialization mechanism to convert the contents of the object into a byte stream that can be sent across the network.
- Both client and server must have a definition for the type being passed (or must be able to get hold of one).
- The remote interface definition uses standard Java syntax to indicate where objects are used as parameters or return values.

When using complex Java types as part of a Web Service, you must address the same issues. However, there is the added complication that you must do this in a platform and language-independent way. Therefore, the following is needed to pass complex parameters as part of a Web Service method:

- Provide a mechanism to marshal and unmarshal the contents of a complex Java type into an XML format that can be used as part of a SOAP message
- Deliver the marshalling and unmarshalling mechanism on both the client and the server
- Indicate in the WSDL definition that certain parameters or return values are complex types, and provide a mapping between the complex types and their associated marshalling/unmarshalling code

Consider also the situation where you are provided with WSDL that has been generated from a non-Java Web Service, such as a Web Service implemented using Microsoft .NET components. This may also contain definitions for complex types that must be mapped into Java types to use that Web Service from Java.

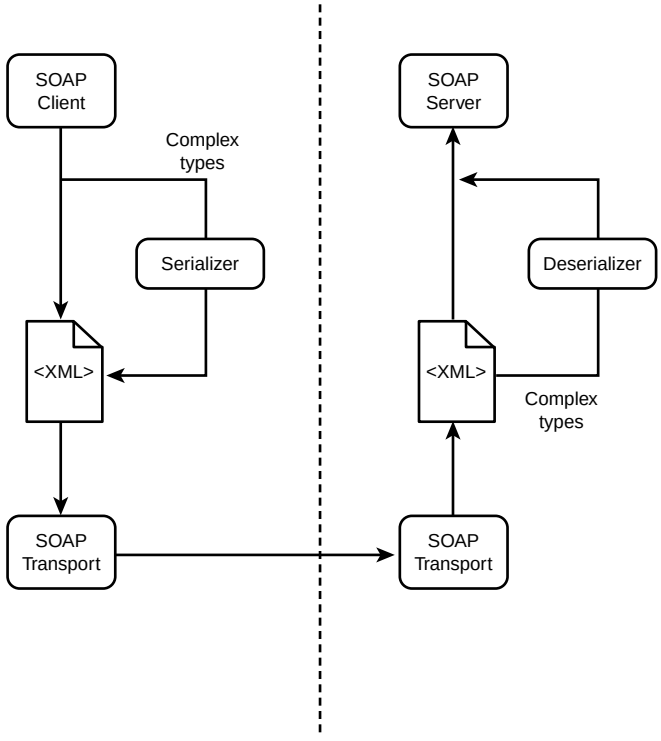
Somebody has to do this mapping, and it is not necessarily straightforward. Sometimes it can be done using automated tools, while at other times it may require custom code.

Mapping Complex Types with Serializers

The JAX-RPC specification defines a serialization framework that can be used to map between complex Java types and their XML representations in WSDL and SOAP. You will define serializer and deserializer classes for each complex type that will be called on by the Web Service runtime to perform this conversion. (See Figure 20.10.)

FIGURE 20.10

The role of serializers and deserializers in a SOAP request.



Serializers fall into three types:

- A set of standard serializers are provided as part of the Java Web Service framework. These include serializers for arrays and common classes, such as `java.util.Date`.
- Given a JavaBean, a generic serializer can use its getters to extract its data when marshalling, and the associated deserializer can use the setters to populate a new instance when unmarshalling.
- A custom serializer can be written to create an XML representation of any Java class.

Obviously, the creation of a custom serializer is not a trivial task. Therefore, it is best to try to keep to standard classes and JavaBeans.

To see how complex type mapping works, consider how you could improve the `SimpleOrderServer` seen previously. The `submitOrder` method took three parameters—`customerId`, `productCode`, and `quantity`. A more realistic service would take a variety of customer information together with a list of line items, each of which would describe a product and the quantity required of that product. The `productCode` and `quantity` can form the basis of a simple line item to start improving the order service.

The `LineItemBean` formed from these is shown in Listing 20.18. As you can see, this provides getters and setters for both the product code and the quantity.

It is important that the JavaBean has a no-argument constructor and the full compliment of getters and setters. Failure to provide all of these can lead to exceptions when trying to marshal or unmarshal these types.

LISTING 20.18 `LineItemBean.java`—Encapsulating the Product Information in a JavaBean

```
 1: package webservices;
 2:
 3: public class LineItemBean
 4: {
 5:   private String _productCode;
 6:   private int _quantity;
 7:
 8:   public LineItemBean(String productCode, int quantity)
 9:   {
10:     _productCode = productCode;
11:     _quantity = quantity;
12:   }
13:
14:   public LineItemBean()
15:   {
16:   }
17:
18:   public String getProductCode()
19:   {
20:     return _productCode;
21:   }
22:
23:   public void setProductCode(String productCode)
24:   {
25:     _productCode = productCode;
26:   }
27:
28:   public int getQuantity()
29:   {
30:     return _quantity;
31:   }
32:
33:   public void setQuantity(int quantity)
34:   {
35:     _quantity = quantity;
36:   }
37: }
```

The server-side code can then be altered to use this JavaBean, as shown in Listing 20.19.

LISTING 20.19 `BeanOrderServer.java`—Using the `LineItemBean`

```
 1: package webservices;
 2:
 3: public class BeanOrderServer
 4: {
 5:   public String submitOrder(String customerID, LineItemBean item)
 6:   {
 7:     String receipt = "Thank you, " + customerID + "\n";
 8:     receipt += "You ordered " + item.getQuantity() + " " +
➥                                    item.getProductCode() + "'s\n";
 9:     receipt += "That will cost you " + (item.getQuantity() * 50) +
➥                                                        " Euros";
10:
11:     return receipt;
12:   }
13: }
```

Before you can deploy this updated class, you need a way to tell the Axis server how it can unmarshal an instance of a `LineItemBean`. If you do not do this, an exception will occur when the server attempts to service a call to `submitOrder`.

Given that the `LineItemBean` is a JavaBean, the Axis `BeanSerializer` can be used to marshal and unmarshal its contents. You instruct the Axis server to use this serializer by defining a bean mapping in the deployment descriptor. Listing 20.20 shows an updated form of the order service deployment descriptor containing a bean mapping between lines 5–7. The bean mapping associates an XML qualified name (a tag name and associated namespace) with a particular Java class. On line 9, the tag `LineItem` from the namespace `urn:com-acme-commerce` is defined as representing the Java class `webservices.LineItemBean`.

LISTING 20.20 Defining a Bean Serializer for Use with the `BeanOrderService`

```
1: <admin:deploy xmlns:admin="AdminService">
2:  <service name="BeanOrderService" pivot="RPCDispatcher">
3:   <option name="className" value="webservices.BeanOrderServer"/>
4:   <option name="methodName" value="submitOrder"/>
5:   <beanMappings>
6:    <acme:LineItem xmlns:acme="urn:com-acme-commerce"
➥             classname="webservices.LineItemBean"/>
7:   </beanMappings>
8:  </service>
9: </admin:deploy>
```

All name-to-class mappings contained within the `<beanMappings>` element will be performed by the `BeanSerializer`. After you have deployed the `BeanOrderService`, a listing of the deployed Axis services will reveal a full type mapping for the bean, as shown in the following (you may need to restart your Web server for this mapping to show up):

```
<service pivot="RPCDispatcher" name="BeanOrderService">
  <option name="methodName" value="submitOrder"/>
  <option name="className" value="webservices.BeanOrderServer"/>
  <typeMappings>
    <typeMapping type="ns:order"
                 xmlns:ns="urn:com-acme-commerce"
                 classname="webservices.LineItemBean"
  deserializerFactory="org.apache.axis.encoding.BeanSerializer$BeanSerFactory"
  serializer="org.apache.axis.encoding.BeanSerializer"/>
  </typeMappings>
</service>
```

You can see that the mapping information provided in the deployment descriptor has been augmented by the class names for the `BeanSerializer`.

The final link-up is made on the server side by indicating which parameters to the Web Service method should be subject to this mapping. The WSDL generated for the service shows that the second parameter (`arg1`) is of the type `nsl:LineItem`. This type associates it with the namespace `urn:com-acme.commerce` as defined by the bean mapping in the deployment descriptor.

```
<definitions
        targetNamespace="http://localhost:8080/axis/services/BeanOrderService"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:serviceNS="http://localhost:8080/axis/services/BeanOrderService"
        xmlns:ns1="urn:com-acme-commerce"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="submitOrderRequest">
    <part name="arg0" type="xsd:string"/>
    <part name="arg1" type="ns1:LineItem"/>
  </message>
  ...
```

The mapping is now set up on the server side. You can now call this service, confident that the correct unmarshalling will take place. All that remains is to call the service from a client. The code shown in Listing 20.21 shows a client for the `BeanOrderService`.

LISTING 20.21 A Client for the `BeanOrderService`

```
1: import java.rmi.RemoteException;
2: import java.net.MalformedURLException;
3:
4: import org.apache.axis.AxisFault;
```

LISTING 20.21 Continued

```
 5: import org.apache.axis.client.ServiceClient;
 6: import org.apache.axis.encoding.BeanSerializer;
 7: import org.apache.axis.utils.Options;
 8: import org.apache.axis.utils.QName;
 9:
10: import webservices.LineItemBean;
11:
12: public class BeanOrderServiceClient
13: {
14:   public static void main(String [] args)
15:   {
16:     String customerId = "unknown";
17:     String productCode = "Widget";
18:     int quantity = 1;
19:     ServiceClient client = null;
20:
21:     try
22:     {
23:       Options options = new Options(args);
24:       client = new ServiceClient(options.getURL());
25:       args = options.getRemainingArgs();
26:     }
27:     catch (MalformedURLException ex)
28:     {
29:       System.err.println("Option error: " + ex);
30:       System.exit(2);
31:     }
32:     catch (AxisFault ex)
33:     {
34:       System.err.println("Option error: " + ex);
35:       System.exit(2);
36:     }
37:
38:     if (args == null || args.length != 3)
39:     {
40:         System.out.println("Usage: SimpleOrderClient -l<endpoint>" +
41:     "<customerId> <productCode> <quantity>");
42:         System.exit(1);
43:     }
44:     else
45:     {
46:         customerId = args[0];
47:         productCode = args[1];
48:         quantity = Integer.parseInt(args[2]);
49:     }
50:
51:     LineItemBean lineItem = new LineItemBean(productCode, quantity);
52:
53:     QName lineItemAssociatedQName = new QName("urn:com-acme-commerce",
➥                                         "LineItem");
```

LISTING 20.21 Continued

```
54:
55:     BeanSerializer serializer =
➥                   new BeanSerializer(webservices.LineItemBean.class);
56:
57:     client.addSerializer(webservices.LineItemBean.class,
58:                          lineItemAssociatedQName,
59:                          serializer);
60:
61:     String receipt;
62:     try {
63:         receipt = (String)client.invoke("BeanOrderService",
64:                                         "submitOrder",
65:                                         new Object[] { customerId, lineItem });
66:     } catch (AxisFault fault) {
67:         receipt = "No receipt";
68:         System.err.println("Error : " + fault.toString());
69:     }
70:
71:     System.out.println(receipt);
72:   }
73: }
```

The main interest begins at line 51 where the `LineItemBean` is instantiated. Following this (line 53), an `org.apache.axis.utils.QName` is created containing the XML qualified name that will represent the `LineItemBean` as it is passed across the network. The value of this `QName` is the one shown in the WSDL generated from the server, namely the tag name `LineItem` in the namespace `urn:com-acme-commerce`.

A `BeanSerializer` is then created that will actually perform the mapping. This serializer must know which particular type of JavaBean it is supposed to map. Consequently, it is given the class type (`webservices.LineItemBean.class`) as a constructor parameter.

Scrolling back to line 24, you can see that the client will be using a `ServiceClient` to invoke the Web Service method. This is initialised with the service URL (`http://localhost:8080/axis/services/BeanOrderService`). On lines 57–59, the serializer and the `QName` are passed to the `addSerializer` method of the `ServiceClient`, together with the JavaBean type, so that the `ServiceClient` knows for which type this serializer is intended.

Finally, the Web Service method itself is invoked on lines 63–65, and the `LineItem` instance is passed as the second parameter. The client-side runtime will then invoke the serializer you have associated with `LineItem` when that parameter is marshalled into the SOAP message.

Going Further with Complex Type Mapping

The use of a `BeanSerializer` is just the start as far as the mapping of complex types is concerned. Taking the example of the `LineItem` further, you would probably pass an array of `LineItem` objects into the method. Alternatively, you could create an `Order` class that held a `Collection` of `LineItems` and pass an instance of `Order` into the method. In this case, you get into the issues of nested serialization and passing complex types into the getters and setters. While some of this is still within the bounds of the serializers provided by Axis, this path leads toward custom serialization—whether such serializers are written by hand or automatically generated by more powerful tools. To create your own serializers, you can subclass the `org.apache.axis.encoding.Deserializer` class, but the creation of custom serializers is beyond the scope of today's work.

Apart from beans, serializers are provided for other standard types, such as `Date` and `Map`, as well as for byte arrays and SOAP arrays.

JAX-RPC defines an extensible type-mapping framework similar to that delivered by Axis in that you can associate serializers with particular Java classes. These mappings are again stored in a type-mapping registry on both the client and the server side. Standard serializers are to be provided for `String`, `Date`, `Calendar`, `BigInteger`, and `BigDecimal`, as well as arrays and JavaBeans.

Another area in which progress should be made for type mapping between Java and XML is the Java API for XML Data Binding (JAXB). JAXB will provide more powerful facilities to map between complex Java types and XML representations of those types. As JAXB progresses, it should provide a useful mapping layer and source of serializers.

Summary

Today, you have seen how Web Services provide a future route for many application integration projects. Web Services provide a framework for the integration of internal or external applications using HTTP and XML. You have seen that Web Services provide a better solution for exposing functionality than existing RPC or Web mechanisms, and you have explored the Web Service functionality offered in Java and J2EE.

You used an existing RPC-style Web Service, both directly using SOAP and through a proxy generated from WSDL. You then created your own server and generated WSDL from this. You looked at how to keep state when using a Web Service and examined the issues around exposing J2EE components as Web Services. Finally, you used complex type mapping to enable a Java object to be passed as a parameter to a Web Service.

Q&A

Review today's material by taking this quiz.

Q **SOAP uses HTTP as a transport, so does this mean that it is restricted to synchronous interaction?**

A Any transport can be used for a SOAP message as long as someone creates a binding for it. SOAP bindings have been defined for SMTP, and such bindings can be created for any other transport mechanism, such as FTP or MQSeries, regardless of whether such mechanisms are synchronous or asynchronous.

Also, although HTTP is inherently synchronous, you can use it to pass XML documents that consist of business "messages" and that form part of a workflow. If the sender of the message is also capable of receiving such messages, it may receive a response of some form at some future point in time. This uses two synchronous interactions to create asynchronous behavior.

Q **Can I use Axis (or JAX-RPC) to send an XML document rather than performing an XML-based RPC call?**

A Although it is possible to send an XML document as a parameter to an RPC call using Axis, document-centric interactions are intended to be serviced by the Java API for XML Messaging (JAXM). You will encounter JAXM in more detail tomorrow.

Q **What sort of information is contained in a WSDL document?**

A A WSDL document contains two basic types of information. It contains the interface for the Web Service that consists of type information, message definitions (parameters and return types), operation definitions (methods that can be called), port types (groupings of methods), and bindings that define how port types are carried over different transports. A WSDL file also contains specific location information for a Web Service in the form of ports that provide a specific location for an instance of a port type and service descriptions that define groups of ports.

Q **What does a `BeanSerializer` do?**

A A `BeanSerializer` lets you use a JavaBean as a parameter or return type in a Web Service method. When sending, the `BeanSerializer` will convert the contents of the JavaBean into XML that can be transported as part of the SOAP message. When receiving, the `BeanSerializer` will unmarshal the XML content for that parameter into an instance of the JavaBean ready to be used by the Java client or server.

Exercises

The intention of this day is for you to create and use RPC-style Web Services. To ensure that you are comfortable with these areas, you should attempt the following tasks.

1. Create a simple stock quote server using a JWS file. The stock quote server should take a string as the stock for which a quote is required (`"SUNW"`, `"MSFT"`, `"ACME"`). The quote service should return a floating point value that reflects the current share price. Your server should just hold two arrays—one of strings and one of associated stock prices (don't get bogged down in the application logic, but feel free to have a bit of fun with the stock prices you return!).
2. Deploy the service and check that it is operational by obtaining its WSDL.
3. Create a client for the service using a generated proxy or a `ServiceClient`. The client should be a simple command-line application that takes a single string that is the name of the stock to retrieve.
4. Convert your JWS service into a standard service that uses a deployment descriptor. Deploy and test your service.
5. Alter your client and server so that the value returned is a JavaBean that contains the name of the stock together with its current price and its 90-day high and low values (so basically, a string and three floating point numbers).

   Use a `BeanSerializer` to marshal and unmarshal your JavaBean. Alter the server's deployment information appropriately. Redeploy the service and test it out.

WEEK 3

DAY 21

Web Service Registries and Message-Style Web Services

Yesterday, you learned about Web Service architecture and you created your own Web Services. Part of the Web Service architecture involved the look up of Web Service information in a registry, but the Web Service information you used was not obtained that way. Also, the Web Services you created were all RPC-style as opposed to document-oriented Web Services. Web Services are more than just a replacement for RPC.

Today, you will

- Investigate the role of Web Service registries
- Examine code that registers and retrieves information using a UDDI registry
- Explore how JAXR will provide uniform access to XML-based registries from Java
- Send and receive SOAP messages using JAXM
- Use a JAXM Provider to send routed SOAP messages

Today's intention is to interact with Web Service registries and to use message-based Web Services (as opposed to RPC-based ones).

Registries for Web Services

As you saw on Day 3, "Naming and Directory Services," naming and directory services are an important part of any distributed environment. J2EE uses JNDI as a way of accessing information about application resources and the location of remote services, such as EJBs and databases.

Because Web Services also operate in a distributed environment, Web Service-oriented naming and directory services are required. In fact, the scope of such services is broader under the Web Services model than under J2EE, because the service will hold organizational and business information as well as Web Service metadata and location information. In Web Service terms, the place to find this information is called a *registry* or *repository*.

What is a Web Service Registry?

As discussed yesterday, a Web Service client can retrieve information about the Web Service it wants to use from a registry. The registry contains information about the service being offered (from currency conversion to the supply of 10-ton trucks). This information includes details about the organization offering the service, the technical access information for the service (interface and transport), and where to find the service.

Tools can be built to browse and search this Web Service registry information. The information retrieved can then be plugged into client applications so that they can integrate with the supplier of the service. This integration can range from fully manual, where a human selects the interface and writes client code to invoke it, to fully automatic, where the selection and invocation are performed by the client application itself.

Why Do I Need One?

To cooperate, applications need information about what services exist and where to find them. You can do this manually by exchanging service definition files, such as WSDL documents. This procedure is reasonably okay when interacting with known business partners. However, manual exchange does have some drawbacks because it does not allow you to find new suppliers of services, and your information must be updated whenever a service location changes.

One alternative is to exchange service information in a partially dynamic way with your business partners by obtaining service definition information from the server on which

the service itself is located. Such a mechanism is described in the Web Service Inspection Language (WSIL) specification. Again, this method of interaction is reasonably okay if you already know where the server is.

To engage in fully dynamic Web Service use, what you really need to do is search for companies or services that match a specific profile (type of business, type of service, location of business, and so on). From this list, you can choose an appropriate service, and then retrieve its technical information with the minimum of fuss. At runtime, it is possible to check the service information again in case the service location information has been updated.

How Do They Work?

There are two basic ways of finding information in a registry. One way is to drill-down, starting at a known location. Under this model, you can iterate through the registry entries below the current location and potentially recurse down through their children or follow links from them to find the information you require. The alternative mechanism is to globally search the registry for the information you require. In this case, you would search based on a particular piece of information about the service you require, such as the organization's name, its line of business, or the interface definition of the service for which you are looking. The use of these three types of search criteria (frequently called white, yellow, and green pages, respectively) provide you with a lot of flexibility for locating the Web Service you need.

What will usually happen is that you will use some form of search to locate an initial registry entry, such as the top-level registry entry describing an organization's business, and then drill down from there to examine the services available.

One issue here is that for searches to be effective, people must agree on ways of describing and categorizing information. This gives rise to classification systems or taxonomies. There are various common taxonomies set up by industry standards bodies to classify information in their particular areas. Such taxonomies can be used as part of the classification of a Web Service in a registry to help clients find the right form of service.

Types of Registry

Web Service registries come in a variety of forms:

- Global registries are publicly available and open to all types of business—similar in concept to the global Domain Name System (DNS) that lets you find anyone's Internet Protocol (IP) assigned to the hostname that forms part of their Uniform Resource Locator (URL).

- Private registries can be set up behind organizational firewalls. These will facilitate the location of Web Services on a company's intranet.
- Site-specific registries will list all of the services offered at the site associated with the registry. For example, `acme.com` could host a registry service that helped you to locate all Web Services provided by `acme.com`.
- Marketplace registries are set up and maintained by a market maker (a third party or industry consortium). Such registries may or may not be publicly available, but will provide some form of selection or quality check on the providers of services before including them in the registry.

There is a good whitepaper about Web Service registry styles on the Web Services part of IBM's DeveloperWorks site at `http://www-106.ibm.com/developerworks/library/ws-rpu1.html`.

Just before leaving the general topic of Web Service registries, it is worth considering how your application—and your business—will actually use Web Services. If you are looking to use Web Services provided by suppliers as part of a supply chain, you may want to use some of the dynamic capabilities of a Web Service registry to locate suppliers and services. However, not all of the information you require will be found dynamically. In real life business terms, you will not necessarily change your supplier of ball bearings from minute to minute. There are other criteria by which you choose your business partners (quality, timeliness, trust, and so on). Therefore, for most applications, the selection of suppliers will usually take place at human speeds and involve humans in the selection. After the selection is made, dynamic lookups can be performed to discover and use the technical service interface and location information.

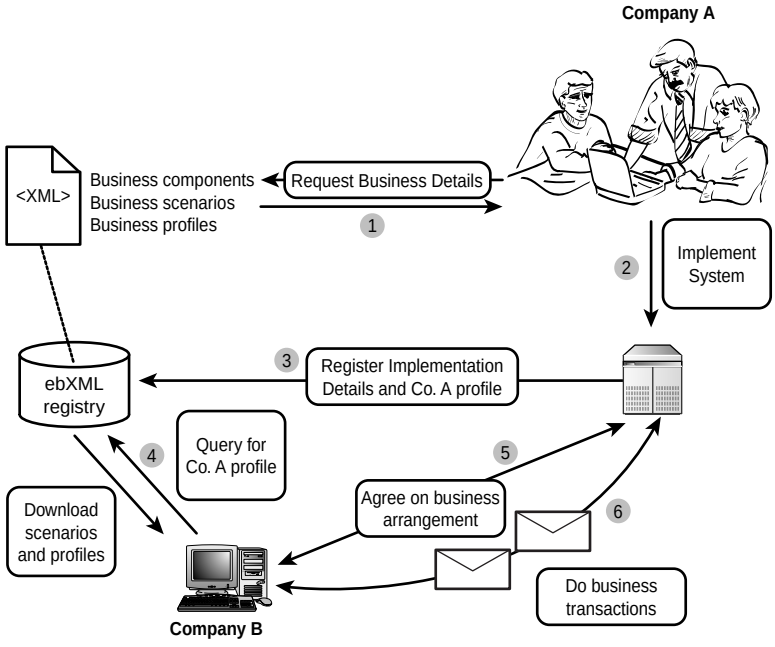
Marketplaces can also help here. A marketplace can provide a qualitative judgement of potential suppliers, making the selection of previously unknown suppliers less risky. This type of qualitative judgement already takes place in various areas of business, such as organizations that provide credit ratings for businesses. Employing such a service helps you to manage your risk when selecting potential business partners. Given such a rating ability, if a marketplace can offer five trustworthy and high-quality suppliers, you can potentially spread the load across these equal suppliers. Even so, this type of dynamic interaction is more likely in information-based areas, such as financial data and transactions, than in the realms of 10-ton truck purchase.

ebXML Registry and Repository

One of the principal types of Web Service registry is the ebXML Registry and Repository (R&R). The ebXML R&R plays a central role in the ebXML model of e-commerce, as shown in Figure 21.1.

FIGURE 21.1

Discovery and negotiation based around an ebXML registry and repository.



The ebXML R&R acts as a storage area for the central business document definitions, or Core Components, that are used in ebXML business messages. An organization (Company A) that wants to advertise its services in an ebXML R&R will base its service descriptions on these Core Components and will then implement the service (shown as steps 1 and 2). Company A and its services will then be registered in the ebXML R&R (step 3), including a Collaboration Partner Profile (CPP) that describes the different bindings for each service and the roles and workflows associated with it.

When Company B searches the registry for a service, it can select Company A's offering and download the service information (step 4). Company B can then use the information in Company A's CPP to determine how it should interact with Company A's services. It can then contact Company A with a suggested Collaboration Partner Agreement (CPA) that details the bindings and service interactions required (step 5). This CPA forms the basis of the contact between the two companies. When a CPA has been agreed on by the two companies, business transactions can begin between the two new partners according to the terms set out in the CPA (step 6). Messages sent between the partners will reference a CPA identifier (CPAid) to indicate of which interaction they are a part.

UDDI Overview

UDDI defines an information model for a registry of Web Service information and a set of SOAP messages for accessing that information. The information contained can be split into two parts:

- *Business information*—This provides the name and contact information for the business. It also provides a list of the categories under which the business should be "registered" to help a user find the business.
- *Service information*—This gives information about which Web Services are offered by the business, their interfaces, and location(s).

At the lowest level, you have the description of a specific Web Service. As you have seen from WSDL, this will contain a description of the service interface and endpoint information telling the user where to find the service. UDDI splits this information between two structures:

- A structure called a `tModel` is used to describe the Web Service's interface, such as the operations, parameters, and data types it uses. Separating such interface information from the service location information means that `tModels` can be stored independently and shared by multiple different services from different businesses. This greatly aids the discovery of compatible service offerings. Although commonly conceptualised as a Web Service description, the information in a `tModel` can describe any mechanism for accessing a service, including e-mail, phone, or fax.

  If you are building a `tModel` from a WSDL interface, you would include the WSDL messages, parameters, return values, complex types, `PortTypes`, and `Bindings`.

- The UDDI `bindingTemplate` structure uses `tModels` to define which particular service is being offered and also provides the location of the service. The location information defines specific endpoints at which the service can be found.

  Again, in WSDL terms, the extra information in the `bindingTemplate` correlates to the WSDL `port` and `service` information.

At a higher level, a `businessService` is used for the logical grouping of services, as defined in `bindingTemplates`. At the top level is a UDDI `businessEntity` that contains the business information. Each `businessEntity` contains one or more `businessServices`.

The UDDI API provides a set of SOAP operations to search for information within this model and to retrieve the relevant parts.

Accessing Information in a UDDI Registry

In this section, you will examine how information in a UDDI registry can be accessed from Java.

The subject of accessing, updating, and searching UDDI-based registries would fill a book in itself. This section is intended to give you a head start in using a UDDI-based registry rather than showing you all the nuts and bolts required.

As a Java developer, you do not really want to have to build SOAP messages to communicate with a UDDI registry. What you want is a Java-based API that hides away the SOAP manipulation. In the rest of this section on registries, you will examine three alternatives:

- *UDDI4J*—IBM's Java implementation of the UDDI API
- *IBM WSTK Client API*—A higher-level API that abstracts some of the UDDI complexity
- *JAXR*—The Java API for XML Registries being developed through the JCP

Manipulating Service Information using UDDI4J

Before you start manipulating data in a registry, you must choose which registry you will use:

- *A public production registry*—This is a "live" business registry, such as those hosted by IBM and Microsoft. You should only manipulate "real" business data in these registries. You can search such a registry freely, but if you want to publish data, you will be required to obtain a login and password.
- *A public test registry*—This is a generally available resource for testing the registration and discovery of Web Services. As with production registries, public test registries are hosted by IBM and Microsoft. If you want to publish data about your test services, you will be required to obtain a login and password.
- *A locally hosted registry*—There are several UDDI registries available that you can configure in your own environment (IBM's downloadable UDDI registry and jUDDI are two examples). These registries can be used internally within your company for Web Service discovery and testing.

Listing 21.1 shows how UDDI4J can be used to register a business in the IBM test registry.

LISTING 21.1 RegisterBusiness.java—A Simple UDDI Client

```
 1: import com.ibm.uddi.*;
 2: import com.ibm.uddi.datatype.business.*;
 3: import com.ibm.uddi.response.*;
 4: import com.ibm.uddi.client.*;
 5: import org.w3c.dom.*;
 6: import java.util.*;
 7: import java.security.*;
 8:
 9: public class RegisterBusiness
10: {
11:   public static void main (String args[])
12:   {
13:     System.setProperty("java.protocol.handler.pkgs",
14:                        "com.ibm.net.ssl.internal.www.protocol");
15:     Security.addProvider(new com.ibm.jsse.JSSEProvider());
16:
17:     UDDIProxy proxy = new UDDIProxy();
18:
19:     try
20:     {
21:       proxy.setInquiryURL(
22:           "http://www-3.ibm.com/services/uddi/testregistry/inquiryapi");

23:       proxy.setPublishURL(https://www-3.ibm.com/services/uddi/ +
24:                                "testregistry/protect/publishapi");
25:
26:       AuthToken token = proxy.get_authToken("fbloggs", "Wibble" );
27:
28:       System.out.println("Authentication Token: " +
➥                                 token.getAuthInfoString());
29:
30:       Vector businessEntities = new Vector();
31:
32:       BusinessEntity businessEntity =
➥                     new BusinessEntity("", "Bloggs Business");
33:       businessEntities.addElement(businessEntity);
34:
35:       BusinessDetail detail =
36:                  proxy.save_business(token.getAuthInfoString(),entities);
37:
38:       businessEntities = detail.getBusinessEntityVector();
39:       BusinessEntity returnedBusinessEntity =
➥                       (BusinessEntity)(businessEntities.elementAt(0));
40:
41:       System.out.println("List businesses starting with B to find ours");
42:
43:       BusinessList list = proxy.find_business("B", null, 0);
44:
45:       Vector businessInfoVector  =
➥                     list.getBusinessInfos().getBusinessInfoVector();
46:       for (int i = 0; i < businessInfoVector.size(); i++)
```

LISTING 21.1 Continued

```
47:       {
48:         BbusinessInfo businessInfo =
➥                     (BusinessInfo)businessInfoVector.elementAt(i);
49:         System.out.println(businessInfo.getNameString());
50:       }
51:     }
52:     catch (UDDIException ex)
53:     {
54:       DispositionReport report = ex.getDispositionReport();
55:       if (report != null)
56:       {
57:         System.out.println("UDDIException" +
58:                            "\n faultCode:" + ex.getFaultCode() +
59:                            "\n operator:" + report.getOperator() +
60:                            "\n generic:"  + report.getGeneric() +
61:                            "\n errno:"    + report.getErrno() +
62:                            "\n errCode:"  + report.getErrCode() +
63:                            "\n errInfoText:" + report.getErrInfoText());
64:       }
65:       ex.printStackTrace();
66:     }
67:     catch (Exception ex)
68:     {
69:       ex.printStackTrace();
70:     }
71:   }
72: }
```

There is no intention to walk through the precise details of the code here. However, you should note the following:

- The UDDI4J API provides a `UDDIProxy` class that acts as a client-side proxy for the UDDI registry. An instance is created on line 17.
- Before using any UDDI-based applications, you must ensure that you have a valid username and password for the registry in question. These are used on line 26 to obtain an authentication token that is passed in subsequent calls to identify this user.
- Because you will need to send your login name and password to authenticate yourself to a public registry, these registries require the use of HTTPS for this purpose. Lines 21–24 show the URLs for inquiries and updates being set on the UDDI proxy. The URL for updates (publish) uses an `https://` protocol identifier. To use this type of URL from a standalone client such as the one shown, you must use the Java Secure Sockets Extension (JSSE). If you are using a pre-JDK 1.4 platform, you will have to download and install this extension. The application shown uses the IBM version of JSSE and initialises it on lines 13–15.

After it has contacted the UDDI registry, the application creates a (very sparse) business entity and publishes this to the registry. It then lists all of the businesses beginning with B to ensure that the update has taken place correctly.

Although at this stage the application is not too complex, the act of retrieving or publishing information can be somewhat tortuous. This is particularly true in the area of service definitions. Consider the situation where you want to publish the WSDL description of your service. To do this, you must convert that service description into a UDDI `tModel`. The UDDI data structure corresponding to the `tModel` must be created as an XML document, and this document must then be uploaded to the registry. Even if you like creating XML documents, there is still the issue of retrieving the relevant parts of the service description from your WSDL to import this into the `tModel` document. Although there is another IBM API, called WSDL4J, that can help you to perform this manipulation, things are becoming fairly messy by now.

Manipulating Service Information Using the IBM WSTK Client API

Because of the complexity associated with using UDDI4J, you may decide to use the IBM WSTK Client API that provides a higher-level abstraction of the UDDI4J API. The WSTK Client API also uses the WSDL4J API to help convert WSDL document information into UDDI `tModel` and service binding information.

The WSTK Client API is centred around the `ServiceRegistryProxy` class, which is an equivalent to the `UDDIProxy`. One way of creating a new `ServiceRegistryProxy` is shown in the following:

```
ServiceRegistryProxy proxy = new ServiceRegistryProxy
➥(inquiryURL, publishURL, userName, password);
```

As you can see, the initial code to use a `ServiceRegistryProxy` instance is little different from that to use a UDDIProxy in that you must provide it with URLs for inquiry and publication, credentials to access the registry, and so forth. However, after the proxy has been initialized, you need essentially only deal with four Java types—service provider, service definition, service implementation, and service interface.

The `ServiceProvider` class represents the business information in your UDDI registry entry. This is a wrapper for the UDDI `businessEntity` information. You will build a `ServiceProvider` object to represent your organization when publishing information, or you will manipulate `ServiceProvider` objects when retrieving business information from the registry.

The `ServiceDefinition` class wraps the UDDI `businessService` structure and provides a way to obtain the interface and location information for the service. The `ServiceImplementation` wraps the UDDI `bindingTemplate` and allows you to retrieve the endpoint information for a service. The `ServiceInterface` class represents the UDDI `tModel` for the service. For both the `ServiceImplementation` and the `ServiceInterface`, a WSDL filename can be passed to the constructor and all of the manipulation of the WSDL document is done for you. This greatly simplifies the creation of service information.

Retrieving and Using Service Information

So far, you have mainly considered the creation of your own business and service information to be registered in the UDDI registry. However, you may also want to use other peoples' services within your application. To do this, you must search the registry and use the information you retrieve.

Regardless of which registry API you use, you will follow the white/yellow/green page model of searching. As an example, the `ServiceRegistryProxy` provides many finder methods, similar to those on an EJB home interface. The finders allow you to specify various facts to help find the required information, such as names, categories, ownership, service definitions, and interface definitions. The information you provide to the finder will depend on what you know already and what type of information you want to get back. The finders will return `ServiceProvider`, `ServiceDefinition`, and `ServiceInterface` objects either singly or in arrays. From these objects, you can find out all you want to know about the service you have found.

The next question is what you do with the service information when you have it. You will probably be retrieving information in one of three scenarios:

- You have an application that uses pre-defined Web Service interfaces. At runtime it will use UDDI to retrieve services that implement that interface and choose the most suitable implementation to use.
- Your application or component forms part of an application creation framework. It will retrieve service information using UDDI and offer the choices to the application builder (think of your typical Java IDE).
- Your application will retrieve service information and dynamically invoke the methods it finds. In this case, the application has no fixed binding to any particular service interface. This is the most complex form of interaction, requiring more coding to invoke the services dynamically and more intelligence to make the right choice about the correct operations and parameters.

As time goes on, standards and tools should make the creation of UDDI-based applications easier. One such standard is the Java API for XML Registries (JAXR) being developed through the Java Community Process.

Using JAXR for Registry Access

The Java API for XML Registries (JAXR) is an attempt to provide a unified model of access to XML-based information stored in registries and repositories. The intention is that this will provide a standard API for such access to ease the development of applications that depend on this information.

At the time of writing, JAXR was still in progress through the JCP. As with any API still in development, some of the calls and mechanisms may change as it matures. Please check the latest JAXR information at the Sun Java Web site (`java.sun.com`).

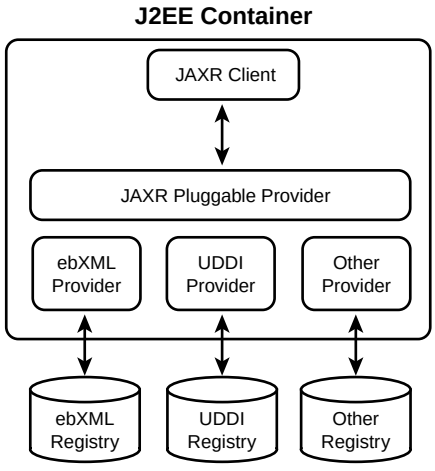
A Generic Approach

The IBM WSTK classes and code you saw in the previous section are specifically targeted at UDDI access. As you know, UDDI is not the only XML-based Web Service registry; the ebXML Registry and Repository also fills this role. In addition, there are various other XML-based registries that are used in different e-commerce frameworks. JAXR intends to provide a set of generic APIs that work on a generic registry information model. Existing registries can then be mapped onto this generic framework, giving developers the ability to access information in different registries using the same API. This should bring the level of consistency to XML-based registry access that JNDI and JDBC have brought to directory services and databases, respectively.

The overall JAXR architecture is shown in Figure 21.2. The client will interact with the JAXR Pluggable Provider layer that acts as a standard interface and rendezvous point. Underneath the Pluggable Provider, a particular registry-specific provider will be used to obtain information from a particular type of registry. This style of architecture has much in common with those of JDBC and JNDI. In fact, it is anticipated that many JAXR providers will, at least initially, be developed as bridges to existing registry providers, in a similar style to the JDBC-ODBC bridge.

The interaction between the JAXR client and the registry service is performed by a set of Java interfaces defined in the `java.xml.registry` package. To begin working with a registry using JAXR, you would use the `Connection`, `RegistryService` and potentially the `RegistryClient` interfaces. Listing 21.2 shows some typical initialization code for an interaction.

FIGURE 21.2
JAXR architecture.



LISTING 21.2 Example JAXR Client Initialization Code (Based on Early JAXR Specification)

```
 1: import javax.xml.registry.*;
 2: ...
 3: public class JAXRClient implements RegistryClient throws JAXRException
 4: {
 5:   public void init(InitialContext ctxt)
 6:   {
 7:     ConnectionFactory factory =
➥              (ConnectionFactory)ctxt.lookup("JAXRConnectionFactory");
 8:
 9:     // Define connection configuration properties
10:     Properties props = new Properties();
11:     props.put("javax.xml.registry.factoryClass",
12:               "com.sun.xml.registry.ConnectionFactory");
13:     props.put("javax.xml.registry.queryManagerURL",
14:               "http://java.sun.com/uddi/inquiry");
15:     props.put("javax.xml.registry.lifeCycleManagerURL",
16:               "http://java.sun.com/uddi/publish");
17:
18:     Connection connection = factory.createConnection(props, this);
19:
20:     Set credentials = new Set();
21:     ...
22:     connection.setCredentials(credentials);
23:     connection.setLocale(locale);
24:     connection.setSynchronous(false);
25:
26:     RegistryService rs = connection.getRegistryService();
27:     ...
28:   }
29:   ...
30: }
```

Obtaining a `Connection` is very much like any other J2EE resource. A factory is obtained through JNDI, and a connection is created from the factory. As with the UDDI code earlier, the URLs for inquiry and publication are provided when creating the connection (lines 13–16). Authentication credentials are also supplied before the registry service is used. As with UDDI, credentials will almost certainly be required to update information in the registry and may even be required for access to information.

Responses to registry queries can be obtained synchronously or asynchronously. In this case, the client will be using the asynchronous style of interaction with the JAXR provider. Hence, it must implement the `RegistryClient` interface that allows the provider to deliver information and errors to the client. Implementation of this callback interface is not required for synchronous interaction because all information and errors are returned directly from the JAXR calls made by the client.

Using JAXR to Store and Retrieve Service Information

JAXR defines its own information model onto which other information models (such as UDDI or ebXML) can be mapped. The JAXR information model contains many familiar interfaces, including the following:

- `Organization` Similar to a UDDI `businessEntity`
- `Service` Similar to the UDDI `businessService` and `tModel`, containing interface and category information for a service
- `ServiceBinding` Similar to the UDDI `bindingTemplate` containing location information for a service instance
- `Concept` Similar to UDDI categories, used for classifying organizations and services

Collections of such interfaces are passed to and returned from JAXR query and manipulation methods.

After the client has obtained a `RegistryService` reference, it can discover the capabilities of the registry service. Capabilities can be generic or business-related. The capabilities are defined in levels and there are currently two levels—0 and 1. Every JAXR Provider must implement the level 0 capabilities. More advanced Providers will also implement the level 1 capabilities. The capability interfaces are

- The `BusinessLifeCycleManager` interface allows you to create, update, and delete `Organizations`, `Services`, `ServiceBindings`, and `Concepts`. Each method returns a `BulkResponse` that contains a collection either of keys identifying individual registry entries or of exceptions indicating that a save or delete operation failed. This is a level 0 capability.

- The `GenericLifeCycleManager` allows you to save or delete any form of registry entry. This is a level 1 capability.
- The `BusinessQueryManager` allows you to find organizations, services, or concepts based on names, concepts, and binding information. The methods offered are very similar to those described earlier for the IBM WSTK `ServiceRegistryProxy`. This is a level 0 capability.
- The `SQLQueryManager` allows you to submit a SQL query that is executed against the registry data. This treats interface types (for example, `Organization`) as if they were database table names. This is a level 1 capability.

Because the principal registries for Web Services are the ebXML Registry and Repository and the UDDI Registry, the JAXR specification defines bindings for UDDI and ebXML. These bindings detail the mappings between the different registry information models.

You have now seen how Web Service information can be stored and retrieved.

Using a Message-Based SOAP Interface

So far, you have examined Web Services largely from an RPC-oriented perspective. For the rest of today, you will examine how message-based Web Services can be used under J2EE.

Message-Style Versus RPC-style

To a large degree, there is little difference between the mechanics of message-style and RPC-style Web Services. Essentially, a message-style Web Service uses a single operation with a single parameter. This single parameter is an XML document to be processed. There is no reason why you could not define an RPC-style interface that takes a single parameter that is an XML document. Under the covers, in SOAP-land, these interfaces would look largely the same. However, there are some differences in application style:

- RPC services offer a related group of operations. There may be some implication of retained state between operations on the same interface.
- Message services tend to work on document-centric interactions, such as a workflow in which each service accepts the document, changes or processes part of it, and then passes it on.

There are pros and cons to both RPC-style services and message-style services. The choice between them will largely depend on how your application works. This choice is the same as between RMI interfaces and JMS messages in non-Web Service J2EE. As with RMI and JMS in J2EE applications, you can mix and match RPC-style and message-style Web Services as required.

Creating a Client

A messaging client must do two things:

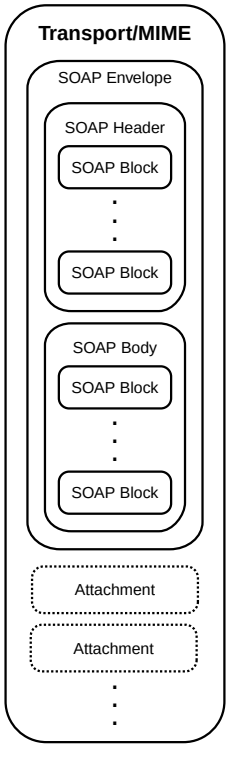
- Create a message to send to the service
- Obtain a connection to the service, or some proxy for it, over which the message can be sent

In terms of a SOAP messaging client, it will send an XML document, possibly with some non-XML attachments. As a developer, you need an API through which to create and populate the underlying SOAP message. A SOAP message has a fixed format, as shown in Figure 21.3, so the API will need to reflect this.

As you can see, the overall message is packaged using MIME to delineate the XML part of the message from the attachments. The XML part of the message is encapsulated in a SOAP envelope. Within the SOAP envelope, there is a SOAP header that contains information relating primarily to the transportation of the message and a SOAP body that holds the principal payload of the message. Both the header and the body consist of XML elements with content, attributes, and namespace information. Your chosen API must allow you to access and populate the contents of a SOAP envelope.

FIGURE 21.3

The contents of a SOAP message.



To send the message, you could create a direct connection to the target service. Otherwise, you could pass it to an intermediary that will route the message to its eventual destination. In both cases, you will need an address for the target service.

You will frequently require some form of reply to your message. The way this is sent depends on the style of interaction. If you send directly to the service, the response could be returned from the call. Otherwise, if the response is to arrive later, you will have to supply your own endpoint information to the service for it to use as the return address for the message. You will also need some mechanism to handle this response when it arrives.

Creating a Service

A message-based service has a single entry point to which messages are delivered. Messages will be received by the service and processed according to their purpose. A synchronous service will generate a response or acknowledgement during the processing and then return this to the sender. An asynchronous service will delegate the message processing to another thread of execution and then return from the call, allowing the sender to proceed. An asynchronous service can send a response to the original sender at a later time.

To receive messages, a service must listen on a named endpoint. In the case of direct, synchronous services, the client will use this to send a message directly to the service. For routed services, the routing system must deliver the message to the target service when it arrives at its destination.

A service can potentially process a single type of message or it could process multiple types. In the latter case, there must be some way for it to determine which type of message has been received to process it correctly. One option is to apply a form of the Command pattern where the message contains information about how it should be processed. It is possible to put such information in the SOAP header, SOAP body, or in an attachment. However, embedding the command in the SOAP body is generally the best way.

Sending and Receiving SOAP Messages with JAXM

The Java API for XML Messaging (JAXM) provides an API for the creation, manipulation, and exchange of SOAP messages. Figure 21.4 shows the overall architecture of a JAXM application.

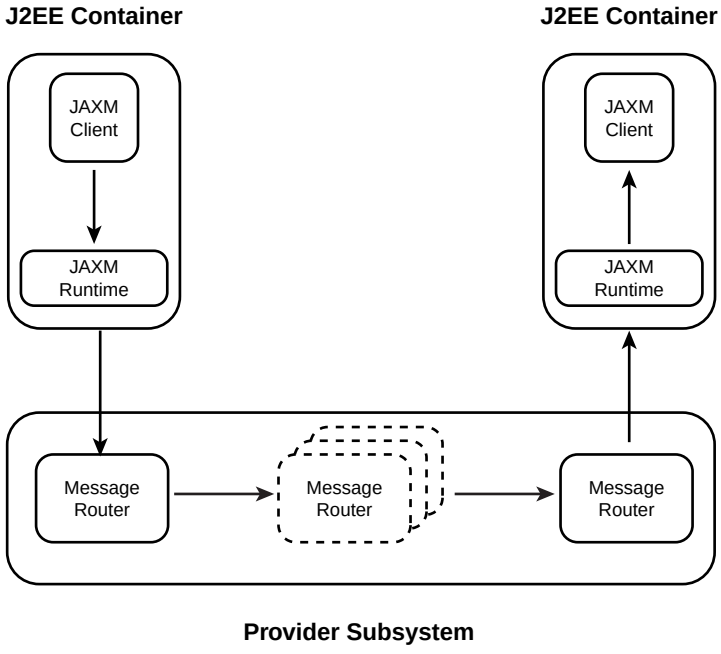


FIGURE 21.4
Overall architecture of a JAXM application.

In a full-blown JAXM application, the message sender connects to a JAXM Provider. The JAXM Provider delivers additional services above and beyond the basic SOAP transport, such as multihop routing and quality-of-service (QoS) guarantees. The sender creates a message and hands it over to the JAXM Provider. The Provider is then responsible for delivering this to the receiver at the given address, possibly via multiple intermediate nodes.

There are two example Providers that come with the JAXM reference implementation:

- *A Provider based on the ebXML Transport, Routing and Packaging (TR&P) specification*—This provides for multi-hop routing and the manipulation of ebXML information in the message, such as the conversation and CPA IDs.
- *A Provider for the SOAP Routing Protocol (SOAPRP)*—Again, this provides for multihop routing and manipulation of SOAPRP-specific fields in the underlying message header.

Because it uses the services of the Provider, a JAXM-based component, whether a sender or receiver of messages, is termed a JAXM client. This nomenclature can make some statements quite confusing, so the terms sender and receiver (or submitter and processor) are used in this section.

The JAXM specification defines five styles of interaction that must be supported between JAXM clients:

- Synchronous information query (response now)
- Asynchronous information query (response later)
- Synchronous updates (acknowledgement now)
- Asynchronous updates (acknowledgement later)
- Logging (no response or acknowledgement)

Full JAXM clients that use a Provider can support all of these models. Indeed, J2EE components that do not use a provider can still send and receive messages using JAXM. In this case, SOAP messages can be delivered directly in a point-to-point fashion. Even a J2SE application can use JAXM to send (but not receive) SOAP messages. Such J2SE-based JAXM clients are referred to as *standalone clients*.

JAXM and J2EE

JAXM is intended to bring message-based Web Services to J2EE and is scheduled to form part of J2EE 1.4. This will allow J2EE components to send or receive SOAP-based messages and act as message-based Web Service clients or servers.

Any J2EE component can be a JAXM client, although servlets and EJBs are the primary focus. In the first release, a base servlet called `JAXMServlet` is provided on which message receivers can be built. The `JAXMServlet` handles the SOAP message in its `doPost` method and delivers a Java `SOAPMessage` object to the receiver. Two types of receiver interface are defined—one for synchronous interaction and one for asynchronous.

Under J2EE, a JAXM client will obtain a suitable Provider through JNDI. Information about the Provider and associated client endpoints will be provided as part of the J2EE container configuration.

Configuring JAXM

JAXM can be installed under either Tomcat or the J2EE RI (or any other J2EE server). Specific instructions are provided in the JAXM documentation (see `jaxm-1.0/docs/jaxm-on-j2ee.html` and `jaxm-1.0/docs/tomcat.html`, respectively). Although the specifics differ, the tasks that need to be done are as follows:

1. Copy the library jars provided with JAXM into a central location (`<J2EE_HOME>/lib/system` or `<CATALINA_HOME>/common/lib`). This includes `jaxm.jar`, `log4j.jar`, `dom4j.jar`, `activation.jar`, `mail.jar`, `jndi.jar`, `client.jar`, `crimson.jar`, `xalan.jar`, and potentially `provider.jar` (out of `provider.war`).

2. For J2EE, you will also need to put these on the `J2EE_CLASSPATH` in the `<J2EE_HOME>/bin/userconfig` script.
3. For Tomcat, you will need to copy `<JAXM-HOME>/jaxm/provider.war` into `<CATALINA_HOME>/webapps`.
4. Start (or restart) the server.

When creating JAXM clients, you will use servlets wrapped up as Web applications. Under Tomcat, you can simply copy these WAR files into the `<CATALINA_HOME>/webapps` directory and restart the server.

To deploy your Web applications under J2EE, you should add them to a J2EE application and then deploy the application (you can use `deploytool` for this).

To ensure that your installation is correct, deploy the `simple.war` WAR from the JAXM `samples` directory. You can then access the following URL (under Tomcat, change the port number for J2EE) to run this simple application and see it confirm a sent and received message:

```
http://localhost:8080/simple/index.html
```

Later, in "Using a JAXM Profile," you examine how the sample JAXM Providers work. This will require you to use the JAXM Provider Admin Tool. To enable this tool, deploy the `provideradmin.war` file from `<JAXM-HOME>/tools`. Under J2EE, you will need to add the `j2ee` user to this Web application as an authorized role.

To run the Provider Admin Tool, use a Web browser to access the URL `http://localhost:8080/provideradmin`. You can log in using a username/password combination of **`tomcat`**/**`tomcat`** under Tomcat and **`j2ee`**/**`j2ee`** under J2EE.

Sending Basic SOAP Messages

Probably the simplest way to use the JAXM SOAP functionality is to send a straightforward SOAP message. You can create a command-line Java application that acts as a standalone JAXM client to create a SOAP message using the JAXM API and send it directly to a SOAP server. Figure 21.5 shows the interaction between a standalone JAXM client and a SOAP server.

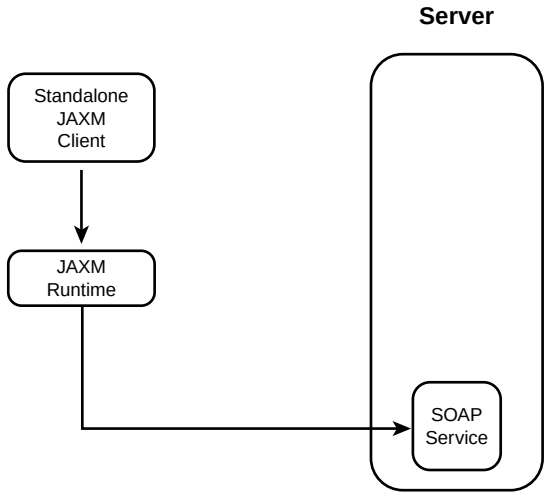


FIGURE 21.5

A standalone JAXM client sending a message to a SOAP service.

The code for such a standalone JAXM client application is shown in Listing 21.3.

LISTING 21.3 `JAXMOrderServiceClient.java`—A Standalone JAXM Client

```
 1: import java.util.Iterator;
 2:
 3: import webservices.LineItem;
 4: import webservices.Order;
 5: import webservices.Receipt;
 6:
 7: import javax.xml.soap.*;
 8:
 9: import javax.xml.messaging.URLEndpoint;
10:
11: public class JAXMOrderServiceClient
12: {
13:   public static void main(String [] args)
14:   {
15:
16:     try
17:     {
18:         MessageFactory messageFactory = MessageFactory.newInstance();
19:         SOAPMessage message = messageFactory.createMessage();
20:
21:         SOAPPart part = message.getSOAPPart();
22:         SOAPEnvelope envelope = part.getEnvelope();
23:
24:         Order order = new Order("Bargain Buys","Strangeways, Manchester");
25:         order.addLineItem(new LineItem("Levi 501", 200));
```

LISTING 21.3 Continued

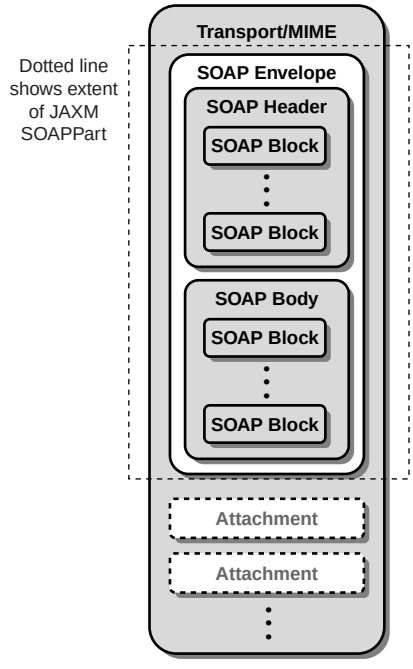
```
        order.addLineItem(new LineItem("Wibble 1000 mp3 player", 33));
        order.addLineItem(new LineItem("Sony Playstation", 10));

        order.marshal(envelope);

        SOAPConnectionFactory connectionFactory =
➥                               SOAPConnectionFactory.newInstance();
        SOAPConnection connection = connectionFactory.createConnection();

        URLEndpoint endPoint =
➥     new URLEndpoint("http://localhost:8080/JAXMOrderService/order");

        SOAPMessage msg = connection.call(message, endPoint);
        SOAPPart p = msg.getSOAPPart();
        SOAPEnvelope env = p.getEnvelope();
        SOAPBody body = env.getBody();

        Receipt receipt = new Receipt();

        Name receiptName = envelope.createName("receipt",
                                               "acme",
                                               "http://acme.com/commerce");

        Iterator iterator = body.getChildElements(receiptName);

        if (iterator.hasNext())
        {
          SOAPBodyElement element = (SOAPBodyElement)iterator.next();
          receipt.unmarshal(element, env);

          System.out.println("Got receipt for order\n\tName:\t\t" +
➥                                          receipt.getName() +
                             "\n\tAddress:\t" + receipt.getAddress() +
                             "\n\tNumber of items:\t" +
➥                                     receipt.getNumItems() +
                             "\n\tCost:\t\t" + receipt.getCost());
        }
        else
        {
          throw new SOAPException(
➥             "receipt SOAPElement is missing from SOAP body");
        }
     }
     catch (SOAPException ex)
     {
       System.err.println("Exception: " + ex);
     }
   }
}
```

The first thing to do is create the SOAP message. Lines 18 and 19 of Listing 21.3 show how a `javax.xml.soap.SOAPMessage` is created from a `javax.xml.soap.MessageFactory`. In this case, a generic SOAP message is required so the factory is created simply through the `newInstance` method. Later, you will obtain a message factory for a specific provider profile.

You can now populate the SOAP message with your information. As shown previously in Figure 21.3, a SOAP message consists of a SOAP part and a set of optional attachments. As a result, you need to retrieve the SOAP part of the message to populate it using the `getSOAPPart` method (line 21). You can then retrieve the SOAP envelope from the SOAP part with the `getEnvelope` method (line 22). The parts of a SOAP message corresponding to the JAXM SOAPPart are shown in Figure 21.6.

FIGURE 21.6

The parts of a SOAP message contained in a JAXM `SOAPPart`.



The SOAP envelope will be populated with the message data. Lines 24–27 show the creation and population of a domain object that represents an order. It contains a name, address, and multiple line items. Each line item contains a product ID and a quantity. The `marshal` method on line 29 asks the `Order` object to populate the given SOAP envelope. This marshalling code will be examined soon, but for now, concentrate on the sending of the message.

After the message is ready, the client needs a `javax.xml.soap.SOAPConnection` to send the message. This is created from a `javax.xml.soap.SOAPConnectionFactory`, as shown in lines 31 and 32. Because the client application is not running inside a J2EE container, the connection factory is created simply through the `newInstance` method. Inside J2EE containers, connection factories can be obtained through JNDI. As with all such resources, if your client is long-running, you should close and release the connection when you are no longer using it (this is not done here because it will be done on application exit).

Because this is a direct SOAP client (that is, it does not use a Provider) you must specify the address of the target server using a `javax.xml.messaging.URLEndpoint`. The endpoint specified in line 34 is the URL of a SOAP server (this happens to be implemented using JAXM, as you will see later, but this is not essential, the server could equally well be implemented in Perl).

You now have the two things you need—a message and somewhere to send it—so you can now send the message (line 36).

Because the service you are calling is a synchronous, request/response service, you will receive a SOAP message as a response. In application terms, this message contains an XML document containing receipt information for the order you have just submitted. To process this receipt, you need to retrieve the XML document from the SOAP message. This XML document will be contained in the SOAP body, so you must retrieve the body from within the SOAP envelope (lines 37–39).

After it has retrieved the SOAP body, the application creates a `Receipt` domain object (line 41) that will represent this information. This object can be passed an XML `<receipt>` element (containing sub-elements for name, address, number of items, and cost) that it will unmarshal and use to populate its data fields. Consequently, the application must find the `<receipt>` element within the SOAP body. To do this, it creates a `javax.xml.soap.Name` based on the SOAP envelope (lines 43–45). This name represents the qualified name (including namespace) of the element to be found—in this case, `(http://acme.com/commerce)receipt`. This is then passed to the `getChildElements` method of the `javax.xml.soap.SOAPBody` (line 47) to return a `java.util.Iterator` that can be used to find all such child elements (there should be only one).

The `javax.xml.soap.SOAPBodyElement` representing the XML `<receipt>` element can then be retrieved and passed to the `Receipt`'s `unmarshal` method that will populate the `Receipt` object from the XML document (lines 51 and 52). The information in the receipt is then printed out (lines 54–57).

Note that during all of this, `javax.xml.soap.SOAPExceptions` may be thrown by the various methods and constructors used. Please refer to the JAXM API documentation for specific details.

Running the Simple Client

To run the client, you will need to deploy the server. Under Tomcat, this is as simple as copying the server WAR file into the `/webapps` directory. The server WAR file (`JAXMOrderService.war`) is provided in Day 21's `examples/JAXMDirect/OrderServer` directory on the CD-ROM.

To run the standalone client, you will need to add to your `CLASSPATH` all the JAR files you placed into `(CATALINA_HOME)/common/lib` earlier.

When you have deployed the server WAR file, start (or re-start) Tomcat and then run the client:

```
prompt> java JAXMOrderServiceClient
Got receipt for order
        Name:            Bargain Buys
        Address:         Strangeways, Manchester
        Number of items: 243
        Cost:            12147.57
```

As you can see, this submits the order as shown in Listing 21.3 to the server and prints out the receipt.

Wow. That seems like quite a lot of work to send a simple message. However, you should now have a good grasp of how it all fits together, which makes the rest of the JAXM API easier to understand.

Populating the Message

So far, you have seen how a message can be created and sent, but the client application did not give details on how a message is populated. As indicated earlier, the JAXM API reflects the SOAP containment hierarchy in that you will obtain a `SOAPPart` from the `SOAPMessage` and then a `SOAPEnvelope` from the `SOAPPart`. Within the `SOAPEnvelope` is a `SOAPBody` and a `SOAPHeader`. The JAXM API provides a DOM-like mechanism for populating and examining the contents of the SOAP body and header.

To examine how this is done, consider the code for the `Order` class shown later in Listing 21.4. The class is essentially a JavaBean that contains a name, address, and a `Vector` to hold the line items. To be encapsulated in the SOAP message, the order should be converted into XML similar to the following:

```
<acme:order xmlns:acme="http://acme.com/commerce">
  <acme:name>Fred Bloggs</acme:name>
  <acme:address>Bury New Road, Manchester</acme:address>
  <acme:items>
```

```
    <acme:item>
      <acme:productId>...</acme:productId>
      <acme:quantity>...</acme:quantity>
    </acme:item>
    ...
  </acme:items>
</acme:order>
```

Recall from the application code in Listing 21.3 that the `marshal` method is passed a SOAP envelope to populate (line 29 of Listing 21.3 and line 16 of Listing 21.4). The SOAP body is retrieved from the envelope, and a new `SOAPBodyElement` is created with the `addBodyElement` method to represent the top level XML `<order>` element (line 24). Note again that qualified names are required when creating new elements (lines 21–23).

Elements to represent the name and address are created below the XML `<order>` element (lines 26–36) by using the `SOAPBodyElement`'s `addChildElement` method. These new elements are of type `SOAPElement`. The text for the elements is added using the `addTextNode` method on `SOAPElement`.

To generate the line items in the XML document, an XML `<items>` element is created in the `SOAPBodyElement`, and this is then populated by iterating through the `Vector` of `LineItem` objects that are asked to marshal themselves underneath the XML `<items>` element (lines 38–49).

LISTING 21.4 `Order.java` Showing the Marshalling and Unmarshalling of XML Data from a SOAP Body

```
 1: package webservices;
 2:
 3: import java.util.*;
 4:
 5: import javax.xml.soap.*;
 6:
 7: public class Order
 8: {
 9:   private Collection _lineItems = new Vector();
10:   private String _name = "";
11:   private String _address = "";
12:
13:   // NORMAL BEAN METHODS AND CONSTRUCTOR REMOVED FOR CLARITY
14:   ...
15:
16:   public void marshal(SOAPEnvelope envelope) throws SOAPException
17:   {
18:     SOAPBody body = envelope.getBody();
19:
```

LISTING 21.4 Continued

```
20:     // Create a new Order element under the body
21:     Name orderName = envelope.createName("order",
22:                                          "acme",
23:                                          "http://acme.com/commerce");
24:     SOAPBodyElement order = body.addBodyElement(orderName);
25:
26:     Name nameName = envelope.createName("name",
27:                                         "acme",
28:                                         "http://acme.com/commerce");
29:     SOAPElement name = order.addChildElement(nameName);
30:     name.addTextNode(name);
31:
32:     Name addressName = envelope.createName("address",
33:                                            "acme",
34:                                            "http://acme.com/commerce");
35:     SOAPElement address = order.addChildElement(addressName);
36:     address.addTextNode(address);
37:
38:     Name itemsName = envelope.createName("items",
39:                                          "acme",
40:                                          "http://acme.com/commerce");
41:     SOAPElement items = order.addChildElement(itemsName);
42:
43:     for (Iterator iterator = _lineItems.iterator();
44:          iterator.hasNext();)
45:     {
46:       LineItem item = (LineItem)iterator.next();
47:
48:       item.marshal(items, envelope);
49:     }
50:    }
51:
52:   public void unmarshal(SOAPBodyElement order, SOAPEnvelope envelope)
➥                                              throws SOAPException
53:   {
54:     Name nameName = envelope.createName("name",
55:                                         "acme",
56:                                         "http://acme.com/commerce");
57:     Name addressName = envelope.createName("address",
58:                                            "acme",
59:                                            "http://acme.com/commerce");
60:     Name itemsName = envelope.createName("items",
61:                                          "acme",
62:                                          "http://acme.com/commerce");
63:
64:     Iterator iterator = order.getChildElements(nameName);
65:
```

LISTING 21.4 Continued

```
66:     if (iterator.hasNext())
67:     {
68:       SOAPElement element = (SOAPElement)iterator.next();
69:       _name = element.getValue();
70:     }
71:     else
72:     {
73:       throw new SOAPException("order SOAPElement is missing name");
74:     }
75:
76:     iterator = order.getChildElements(addressName);
77:
78:     if (iterator.hasNext())
79:     {
80:       SOAPElement element = (SOAPElement)iterator.next();
81:       _address = element.getValue();
82:     }
83:     else
84:     {
85:       throw new SOAPException("order SOAPElement is missing address");
86:     }
87:
88:     iterator = order.getChildElements(itemsName);
89:
90:     if (iterator.hasNext())
91:     {
92:       SOAPElement element = (SOAPElement)iterator.next();
93:
94:       iterator = element.getChildElements();
95:
96:       while (iterator.hasNext())
97:       {
98:         SOAPElement elem = (SOAPElement)iterator.next();
99:
100:        LineItem item = new LineItem();
101:
102:        item.unmarshal(elem, envelope);
103:
104:        addLineItem(item);
105:      }
106:    }
107:    else
108:    {
109:      throw new SOAPException("order SOAPElement is missing items");
110:    }
111:  }
112: }
```

The `unmarshal` method (line 52) used by the server contains the reverse code from the `marshal` method. The `unmarshal` method is passed a `SOAPBodyElement` representing an XML `<order>` element. The `getChildElements` method can then be used to retrieve the XML `<name>`, `<address>`, and `<items>` elements in turn. The method returns an `Iterator` that can be used to retrieve the `SOAPElement` itself (as shown in lines 66–68). The text contents of the `SOAPElement` can be retrieved with the `getValue` method (line 69). As each line item element is retrieved, it is passed to an instance of the `LineItem` class to be unmarshalled (94–105).

The marshalling and unmarshalling code for the `Receipt` and `LineItem` classes is very similar.

Although this is a perfectly adequate way of populating a SOAP message, it is somewhat unwieldy, especially if you already have the information as a Java object or DOM instance. Hopefully, simpler ways of populating JAXM messages will emerge over time, which may include use of the Java API for XML Data Binding (JAXB).

Headers and Attachments

As well as XML information in the SOAP body, JAMX allows you to add and retrieve header information and attachments.

The `SOAPHeader` can be obtained from the `SOAPEnvelope` with the `getHeader` method. If you are creating a message, you can populate the header using the `addHeaderElement` method that returns a `SOAPHeaderElement`. You can then add text or attributes to this header element, as shown in the following:

```
Name txName = envelope.createName("TransactionId", "acme",
                                  "http://acme.com/transactions");
SOAPHeaderElement headerElement = header.addHeaderElement(txName);
headerElement.addTextNode("78d2892ea8af625323c7");
```

There are also specific methods for associating a particular SOAP actor or the SOAP `mustUnderstand` attribute to a header element.

When receiving a message, an `Iterator` that iterates over the header elements can be retrieved with the `examineHeaderElements` method.

Additionally, you may want to add attachments to your SOAP messages by using `SOAPMessage`'s `addAttachmentPart` method. `SOAPMessage` also has a `createAttachment` method that allows you to create an `AttachmentPart` object to attach to a message.

You can provide the data for your attachment directly or through a content handler (part of the JavaBeans Activation Framework—JAF). These attachments can be any form of data, so the key thing is to set the appropriate MIME type. If providing the data directly, you must specify the MIME type when creating the attachment. When using a content handler, this can supply the correct MIME type to the `AttachmentPart` object.

The code required to create an attachment from an image at a given URL is shown in the following code:

```
SOAPMessage message = ...
...
URL url = new URL("http://acme.com/acme_logo.jpg");
AttachmentPart attachment = message.createAttachmentPart(new DataHandler(url));
message.addAttachmentPart(attachment);
```

Receiving a SOAP Message Using JAXM

The SOAP service used by the standalone client you ran previously is implemented using JAXM. The service takes the form of a servlet that is a JAXM client. The servlet runs inside a servlet container and interacts with JAXM, as shown in Figure 21.7. The servlet consumes the SOAP message from the client containing the order, processes it, and returns a SOAP message containing a receipt.

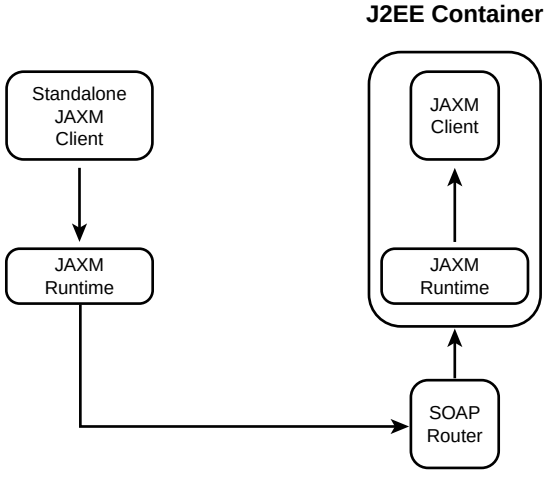


FIGURE 21.7
A JAXM client acting as a SOAP service.

To implement a simple JAXM client is reasonably straightforward. The code for the service used is shown in Listing 21.5. The first thing to note is that the JAXM client implements the `javax.xml.messaging.ReqRespListener` interface (line 8). This interface defines a single method—`onMessage`—that takes a single `SOAPMessage` as a parameter and returns a `SOAPMessage` as a parameter (line 10). If a JAXM client intends to consume SOAP messages, it must implement either `ReqRespListener` or `OnewayListener`, which defines the same `onMessage` method but with no return value.

LISTING 21.5 JAXMOrderServer.java—A Simple JAXM Client Providing a SOAP Order Service

```
 1: package webservices;
 2:
 3: import java.util.Iterator;
 4: import javax.xml.soap.*;
 5: import javax.xml.messaging.JAXMServlet;
 6: import javax.xml.messaging.ReqRespListener;
 7:
 8: public class JAXMOrderServer extends JAXMServlet implements ReqRespListener
 9: {
10:     public SOAPMessage onMessage(SOAPMessage message)
11:     {
12:         try
13:         {
14:             SOAPEnvelope envelope = message.getSOAPPart().getEnvelope();
15:             SOAPBody body = envelope.getBody();
16:
17:             Name orderName = envelope.createName("order",
18:                                                  "acme",
19:                                                "http://acme.com/commerce");
20:
21:             Iterator iterator = body.getChildElements(orderName);
22:
23:             Order order = new Order();
24:
25:             if (iterator.hasNext())
26:             {
27:               SOAPBodyElement element = (SOAPBodyElement)iterator.next();
28:               order.unmarshal(element, envelope);
29:             }
30:             else
31:             {
32:               throw new SOAPException(
➥                "order SOAPElement is missing from SOAP body");
33:             }
34:
35:             System.out.println("Got order from " + order.getName());
36:
37:             Receipt receipt = new Receipt(order.getName(),
➥                                           order.getAddress(),
38:                                  order.getNumItems(), calculateCost(order));
39:
40:             MessageFactory fac = MessageFactory.newInstance();
41:
42:             SOAPMessage msg = fac.createMessage();
43:
```

LISTING 21.5 Continued

```
44:             SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
45:
46:             Name receiptName = envelope.createName("receipt",
47:                                                    "acme",
48:                                                "http://acme.com/commerce");
49:
50:             SOAPBodyElement rcpt =
➥                           env.getBody().addBodyElement(receiptName);
51:
52:             receipt.marshal(rcpt, env);
53:
54:             return msg;
55:         }
56:         catch(Exception ex)
57:         {
58:             System.err.println(
➥              "Error in processing or replying to a message: " + ex);
59:             return null;
60:         }
61:   }
62:
63:   private double calculateCost(Order order)
64:   {
65:     // Sale now on - everything for $49.99
66:     return order.getNumItems() * 49.99;
67:   }
68: }
```

You will know from yesterday that HTTP-based SOAP messages are delivered to the `doPost` method of a servlet. The question then arises of where this SOAP message is delivered when the submitter sends it. The answer lies in the fact that the `JAXMOrderServer` extends `javax.xml.messaging.JAXMServlet`. The `doPost` method of `JAMXServlet` processes the inbound SOAP message and calls the `onMessage` method defined by its subclass (the signature of the method called depends on which interface the subclass implements). The `JAXMServlet` also takes responsibility for converting any returned `SOAPMessage` into an appropriate HTTP/XML document and sending this back to the caller.

The code that processes the message should be fairly familiar from the standalone client. The servlet extracts the XML `<order>` element from the received SOAP message and then passes it to a Java `Order` object to unmarshal it. Based on the contents of this order, the servlet generates a `Receipt` object to pass back. The servlet creates a `SOAPMessage` and recovers its `SOAPBody` into which the receipt is marshalled.

To compile and deploy your JAXM service, you will need to do the following:

1. Include `jaxm.jar` on your `CLASSPATH` when compiling.
2. Create the correct directory structure for a Web application, including a `WEB-INF` directory and a `web.xml` file (see Day 12, "Servlets," for more detail). The `web.xml` file should define a servlet mapping for the URL by which your client expects to access the service. Place your class files under the `WEB-INF/classes` directory.
3. JAR the Web application into a WAR file and copy the WAR into `(CATALINA_HOME)/webapps`. Start (or restart) Tomcat and then run the client as before.

Note that the example used in this section has a standalone client submitting the order. It is equally possible for the submitter to be a JAXM client itself. This scenario is shown in Figure 21.8.

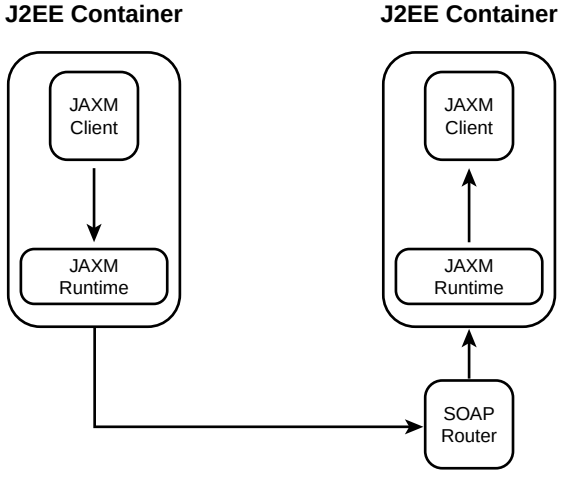


FIGURE 21.8
A JAXM client acting as a SOAP client.

Using a JAXM Profile

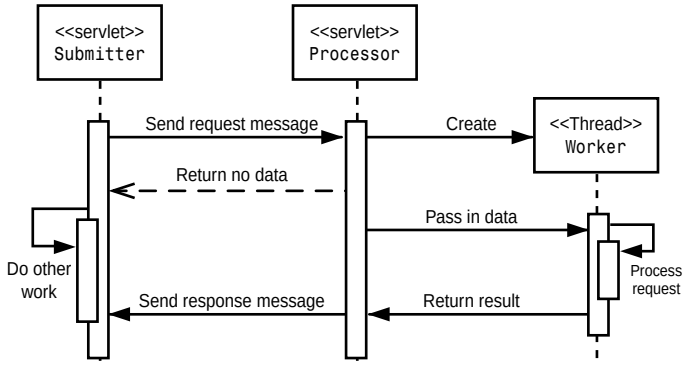
As noted earlier, synchronous, point-to-point message exchange is only one of the intended modes of operation for a JAXM application. To conclude this three-week tour of J2EE past, present, and future, this final section looks briefly at how you use a JAXM Provider. Recall that a Provider delivers extra messaging capabilities, such as multihop routing and quality-of-service guarantees.

The order submission application can be updated to use a Provider. The updated exchange is described in the following:

- The processor JAXM client implements `OnewayListener` through which it will receive orders to process.
- The submitter JAXM client will send a message to the processor and then continue processing without requiring a receipt to be returned.
- The processor will receive a message and then process it. In reality, this would probably involve spawning a thread to process the message while the `onMessage` method returns.

If you wanted to send a receipt, the submitter would also have to implement `OnewayListener`. The processor could then send the receipt back to the submitter and the submitter would receive the receipt through its own `onMessage` method. The full exchange of order and receipt is shown in Figure 21.9. Because the sending of a receipt is the reverse of sending the order, we will just examine how to pass messages in one direction.

FIGURE 21.9
An asynchronous exchange between two JAXM clients.



The JAXM Provider requires additional configuration across several files and tools, which can be tricky to get right. You should ensure that you are fully comfortable with the concepts around the provider before making the changes required to deploy a provider-based JAXM client.

Because the application logic of the JAXM servlets is essentially the same (the creation and consumption of orders), we will concentrate on describing code that is different.

The updated application consists of an order submitter (`SubmittingServlet.java`) and an order processor (`ProcessingServlet.java`). The classes involved are available under the `JAXMProvider` directory in the Day 21 examples code on the CD-ROM.

Sending a Message Using a JAXM Profile

The first thing that the submitter must do if it is to use a Provider is to get a connection to the Provider. This is done in the servlet's `init` method, as shown in Listing 21.6 (lines 27–41). A Provider can also be configured in the servlet's deployment information and obtained using JNDI.

LISTING 21.6 `SubmittingServlet.java`—A JAXM Client That Uses a Provider

```
 1: package soaprp.submitter;
 2:
 3: import java.net.*;
 4: import java.io.*;
 5:
 6: import javax.servlet.http.*;
 7: import javax.servlet.*;
 8:
 9: import javax.xml.messaging.*;
10: import javax.xml.soap.*;
11:
12: import javax.activation.*;
13: import com.sun.xml.messaging.soaprp.*;
14:
15: public class SubmittingServlet extends HttpServlet
16: {
17:   private String submitter ="http://www.acme.com/orderprocessor";
18:   private String processor = "http://www.acme.com/orderprocessor";
19:
20:   private ProviderConnectionFactory cFactory;
21:   private ProviderConnection connection;
22:   private MessageFactory mFactory;
23:
24:   private static final String providerURI =
25:                               "http://java.sun.com/xml/jaxm/provider";
26:
27:   public void init(ServletConfig config) throws ServletException
28:   {
29:     super.init(config);
30:
31:     try
32:     {
33:       cFactory = ProviderConnectionFactory.newInstance();
34:       connection = cFactory.createConnection();
35:     }
36:     catch (Exception ex)
37:     {
38:       System.err.println("Unable to open connection to the provider" +
39:                                                          ex.getMessage());
40:     }
```

LISTING 21.6 Continued

```
41:   }
42:
43:   public void doGet(HttpServletRequest request,
44:                     HttpServletResponse response) throws ServletException
45:   {
46:     try
47:     {
48:       if (mFactory == null)
49:       {
50:         ProviderMetaData metaData = connection.getMetaData();
51:         String[] profiles = metaData.getSupportedProfiles();
52:         String profile = null;
53:
54:         for (int i=0; i < profiles.length; i++)
55:         {
56:           if (profiles[i].equals("soaprp"))
57:           {
58:             profile = profiles[i];
59:             break;
60:           }
61:         }
62:         mFactory = connection.createMessageFactory(profile);
63:       }
64:
65:       SOAPRPMessageImpl message =
66:                             (SOAPRPMessageImpl)mFactory.createMessage();
67:
68:       message.setFrom(new Endpoint(submitter));
69:       message.setTo(new Endpoint(processor));
70:
71:       String order = "http://localhost:8080/jaxm-soaprp-order/orders/" +
72:                                   "order1.xml";
73:       URL orderDocument = new URL(order);
74:       DataHandler dh = new DataHandler(orderDocument);
75:
76:       AttachmentPart attachment = message.createAttachmentPart(dh);
77:       attachment.setContentType("text/xml");
78:       message.addAttachmentPart(attachment);
79:
80:       System.out.println("SubmittingServlet: doGet: Sending message");
81:
82:       connection.send(message);
83:
84:       System.out.println("SubmittingServlet: doGet: Sent message");
85:
86:       PrintWriter writer = response.getWriter();
87:       writer.println("<html><body>Looking good...</body></html>");
```

LISTING 21.6 Continued

```
88:       writer.flush();
89:       writer.close();
90:     }
91:     catch (Exception ex)
92:     {
93:       System.err.println("SubmittingServlet: doGet: " + ex.getMessage());
94:     }
95:   }
96: }
```

The `doGet` method handles the submit request (line 43) as follows:

- Obtain a message factory for your chosen profile. The code (lines 48–63) looks through the profiles available and selects the SOAP routing protocol—`soaprp`.
- Create a `SOAPMessage` representing a fixed order to send. This takes the form of a `SOAPRPMessageImpl`, which is a concrete form of `SOAPMessage`.
- Populate and send the message (lines 68–82). In this case, for simplicity, a pre-built order, `order1.xml`, is loaded into an attachment of this message. Because this is a routed SOAP message, you must set the source and destination addresses using two endpoint URIs that are discussed later. If you were using the ebXML profile, you could set ebXML-specific message fields, such as the CPAid, and the sender/receiver parties. Note also that no response is expected from the message processor.
- Displaying an HTML page to the user stating that the order has been sent (line 87).

Do not try to obtain a message factory in the servlet's `init` because this can hang the servlet engine.

Receiving a Message Using a JAXM Profile

The submitter sets the destination address on the SOAPRP message in the form of endpoints. The processor is registered with the SOAPRP Provider under the URI `http://www.acme.com/orderprocessor`. The Provider must map this address to an actual servlet URL to deliver messages to this address. This mapping information is contained in an XML file—`client.xml`—that forms part of each JAXM client's Web application and is stored in the `WEB-INF/classes` directory.

To use the Provider, you must also configure router mappings for the endpoints. These mappings indicate to which router JAXM should pass messages when they are sent to the processor URI. To configure the provider under Tomcat, point your Web browser at the URL `http://localhost:8080/provideradmin/index.jsp`. Here you will be able to configure the SOAPRP mappings for HTTP-based JAXM clients by expanding the menu on the left side as follows: JAXM Provider, Profiles, SOAPRP, HTTP. You can now select Create New Endpoint Mapping from the Available Actions drop-down list box and set up the following router mapping:

```
URI: http://www.acme.com/orderprocessor
URL: http://localhost:8080/jaxm-provider/receiver/soaprp
```

This mapping points to the message receiver for SOAPRP messages (a servlet located at `/provider/receiver/soaprp`), which is part of the SOAPRP Provider. This message receiver acts as the local router for such messages. When the client servlet is initialized and creates a new connection, its endpoint-to-URL mapping is registered with the SOAPRP Provider. The Provider builds a routing table based on all of the registered endpoints. This routing table is used to route all JAXM-based SOAPRP services that arrive at the local host. When a message arrives at the message receiver, the message receiver consults this routing table to dispatch the message. If this were a full-blown application and deployment, this SOAPRP Provider may figure out that the message is not for a local service and so forward it on to the next router. Alternatively, the endpoint mapping for a particular URI can cause a locally-sent message to be delivered to a different router (possibly an "edge" router for this organization).

The order processor, shown in Listing 21.7, implements the `OnewayListener` interface. The order processor obtains a Provider connection, waits for a message and processes it when it arrives. In this simple case, the processing just consists of logging the message to standard output. As you can see, there are very few requirements on JAXM message recipients.

LISTING 21.7 Processing`Servlet.java`—A JAXM Client That Receives Messages from a Provider

```
 1: package soaprp.processor;
 2:
 3: import javax.xml.messaging.*;
 4: import javax.xml.soap.*;
 5: import javax.servlet.*;
 6: import javax.servlet.http.*;
 7: import com.sun.xml.messaging.soaprp.*;
 8:
 9: public class ProcessingServlet extends JAXMServlet
10:                                              implements OnewayListener
11: {
```

LISTING 21.7 Continued

```
 12:   private ProviderConnectionFactory factory;
 13:   private ProviderConnection connection;
 14:   private static final String providerURI =
 15:                                "http://java.sun.com/xml/jaxm/provider";
 16:
 17:   public void init(ServletConfig servletConfig) throws ServletException
 18:   {
 19:     super.init(servletConfig);
 20:
 21:     try
 22:     {
 23:       factory = ProviderConnectionFactory.newInstance();
 24:       connection = factory.createConnection();
 25:       setMessageFactory(new SOAPRPMessageFactoryImpl());
 26:     }
 27:     catch (Exception ex)
 28:     {
 29:       throw new ServletException("ProcessingServlet: init: " +
 30:                                                    ex.getMessage());
 31:     }
 32:   }
 33:
 34:   public void onMessage(SOAPMessage message)
 35:   {
 36:     System.out.println("ProcessingServlet: onMessage: Received message:");
 37:     try
 38:     {
 39:       message.saveChanges();
 40:       // Just log the message for now...
 41:       message.writeTo(System.out);
 42:     }
 43:     catch(Exception ex)
 44:     {
 45:       System.err.println("ProcessingServlet: onMessage: " +
ex.getMessage());
 46:     }
 47:   }
 48: }
```

Should you decide to try out the JAXM Provider-based code shown here (and provided on the CD-ROM), you may find that on early versions of the reference implementation, the initial message gets "stuck" and is not delivered to the order processor. If this happens, try sending another message (just use the browser Back button and click the hyperlink again), which has the effect of flushing out the initial message.

That is a submitter and processor for JAXM using a Provider. The classes and Web applications are quite similar in many respects for both submitter and processor because they are both essentially just clients of the JAXM provider taking on the alternate roles of client and server in this particular case.

Summary

Today, you have seen how registries play a vital role in allowing Web Service clients to locate and use Web Services. UDDI provides a basic interface for such interactions, but JAXR provides a higher-level interface that makes interaction easier and also allows you to interact with different types of XML-based registry.

You sent SOAP messages between a JAXM client and server. You then improved this simple exchange by using a JAXM Provider that can deliver multihop routing and other value-added services.

Q&A

Review today's material by taking this quiz.

Q Why are different categories of information required in a UDDI registry?

A When you come to search for Web Services, you may know only certain things about them, such as the name of the company that offers the service or the type of interface you expect from the service. By providing business (white pages), category (yellow pages), and technical (green pages) information, UDDI provides a lot of flexibility in the way that you can search for Web Services.

Q Do you have to use a JAXM Provider to take advantage of all the different types of interaction between JAXM clients?

A No. The use of a Provider adds robustness and extra functionality but is not essential as long as endpoints are defined for both parties in the interaction. All of the following JAXM interactions can be performed without the use of a JAXM Provider:

- Synchronous information query (response now)
- Asynchronous information query (response later)
- Synchronous updates (acknowledgement now)
- Asynchronous updated (acknowledgement later)
- Logging (no response or acknowledgement)

Q Why might you use IBM's WSTK Client API to access registry information in preference to UDDI4J?

A The WSTK Client API provides a higher-level interface to UDDI and WSDL. For example, the developer can use the WSTK Client API to decompose WSDL documents into the equivalent UDDI objects rather than having to perform such low-level manipulation themselves. As such, it is easier to work with XML-based registry information using the WSTK Client API than it is through lower-level APIs such as UDDI4J and WSDL4J.

Q What does it mean to be a JAXM client?

A A JAXM client is any J2EE component or J2SE application that uses the JAXM API to send or receive SOAP messages. A full JAXM client will use a Provider to send messages and will register with a Provider to receive messages.

Exercises

You have covered many topics today. One of the most common tasks that you will want to perform is to send XML documents as SOAP messages.

1. Create a JAXM client that accepts a SOAP message representing a new customer for the Job Agency and returns a response containing an assigned login. Generate your own string for the assigned login; there is no need to integrate this client with the agency EJBs you created earlier.

   The expected XML message in the SOAP body should be as follows:

```
<acme:customer xmlns:acme="http://acme.com/commerce">
  <acme:name>Fred Bloggs</acme:name>
  <acme:email>fred@bloggs.org</acme:email>
  <acme:address>
    <acme:addressLine>Bury Old Road</acme:addressLine>
    <acme:addressLine>Manchester</acme:addressLine>
    <acme:addressLine>M25 7ZZ</acme:addressLine>
  </acme:address>
</acme:customer>
```

   The generated XML response body should look like the following:

```
<acme:login xmlns:acme="http://acme.com/commerce">
  Fred Bloggs1234
</acme:login>
```

   An example solution is available under the `JobService` directory in the Day 21 exercise code on the CD-ROM.

2. Create a JAXM client servlet that submits a new customer to the service you created in step 1. This should generate a SOAP message containing the customer information in its SOAP body and then send this to the SOAP service. Send the message directly to the service, do *not* use a JAXM Provider in this instance. Display the assigned login returned from the call.

   An example solution is available under the `JobPortal` directory in the Day 21 exercise code on the CD-ROM.

3. Run the submitter and receiver to test that they work together.

APPENDIX A

An Introduction to UML

A number of the lessons in this book use Unified Modelling Language (UML) diagrams to illustrate the relationships between classes, the interactions between objects, and other types of relationships and events. If you don't know the UML, this appendix offers an accelerated introduction to the UML and the essentials of creating models using the UML. Alternatively, if you do know the UML, you will find this appendix offers a quick reference to the most commonly used elements of the language—the ones used throughout this book.

Introducing the UML

The UML is an Object Modelling Group (OMG) standard that acts as a means of communication in the object-oriented (OO) arena. It defines a notation that consists of graphical elements you can use in UML models. In other words, the notation is the syntax of the modelling language. A meta-model defines the actual notation, and this is normally represented as a class diagram. The meta-model can help you ensure that your models are well formed—that is, that they are syntactically correct.

You can discover more about the UML and the work of the OMG by visiting `http://www.uml.org`.

Although the UML standard defines how you represent items, it does not stipulate the process you use to create models. In other words, the UML does not dictate what steps you must take to create a model; however, processes do exist that you can use, such as the Rational Unified Process (RUP). Alternatively, you can use your own process, which implements parts of the UML, that suit your particular needs.

You can learn about the RUP and how it relates to the UML by visiting `http://www.rational.com`.

At this point, you might wonder why you should take the time to learn and use the UML. There are a number of very good reasons to use the UML, including the following:

- It is less verbose than ideas communicated through natural language.
- It is quicker to write and modify than ideas expressed in code.
- It is independent of programming languages, so people can understand it regardless of their programming backgrounds.
- It aids good OO design and helps exploit the features of OO programming languages.
- It is a universal standard, so anyone that learns it can understand models created by others.

In addition to these advantages, different people can use the UML in different scenarios. This appendix will often discuss UML models in three different scenarios or levels, which were originally suggested by Fowler and Scott (*UML Distilled, Second Edition*: 1999):

- *Conceptual level*—Looks at the domain in terms of concepts, with no regard to actual software
- *Specification level*—Looks at the software in terms of its interfaces
- *Implementation level*—Looks at the actual classes of the software

Working at a conceptual level allows you to develop and modify models very quickly. Typically, you can work at this level and then progress your models to either the specification level or implementation level. In practice, it is generally better to work at the

specification level before the implementation level, so that you end up coding classes to interfaces rather than the other way around. Remember though, that these are simply suggestions because the UML does not define a rigid process for creating models.

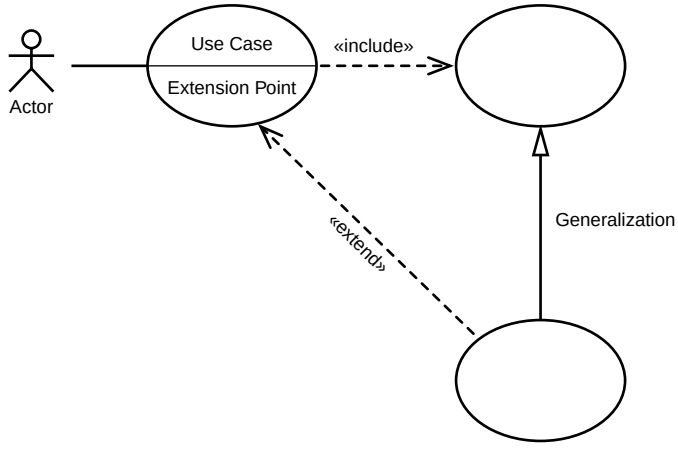
Use Case Diagrams

A Use Case Diagram allows you to model the interactions between a user and a system. A Use Case consists of one or more scenarios, where a scenario is a sequence of steps taken when a user interacts with a system. The boundaries that define Use Cases and scenarios often blur, so you might say that a certain group of interactions form a single Use Case, but another person might group those interactions into more than one Use Case. Either approach is correct, because it is up to the modeller to use his or her discretion when deciding what forms a Use Case.

The notation for Use Cases is quite straightforward, as Figure A.1 shows.

FIGURE A.1

Use Case notation.

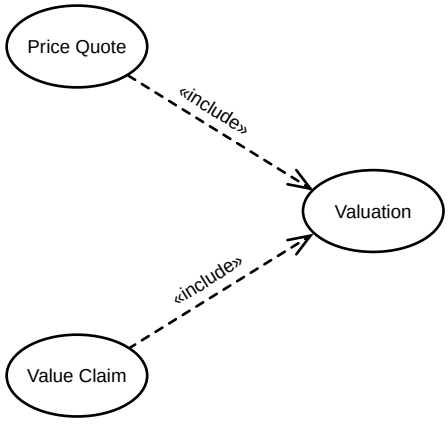


The *actor* is an individual or other system that interacts with the system. As you can see in Figure A.1, the actor carries out Use Cases. A single Use Case can have several actors or a single actor. Likewise, a single actor can perform several Use Cases.

The `<<include>>` notation avoids repetition, because it allows you to copy the behavior of one or more Use Cases. For example, a motor vehicle insurance company's system can allow an actor to get a price quote or make a claim. In both instances, a valuation is required for the Use Case. Figure A.2 shows the two Use Cases, including the valuation Use Case, thus avoiding repetition.

FIGURE A.2

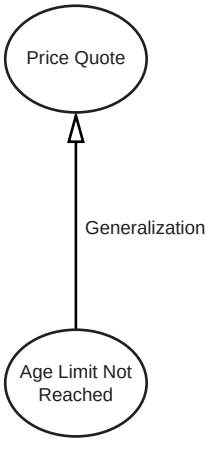
The `<<include>>` notation.



The `generalization` notation that Figure A.1 shows allows you to describe a variation on the normal course of behaviour for a Use Case. For example, the price quote Use Case illustrated previously might have an alternative that the system uses if the proposed insured is classified as a high risk, perhaps because he or she is young. Figure A.3 shows the use of the generalization notation in this instance.

FIGURE A.3

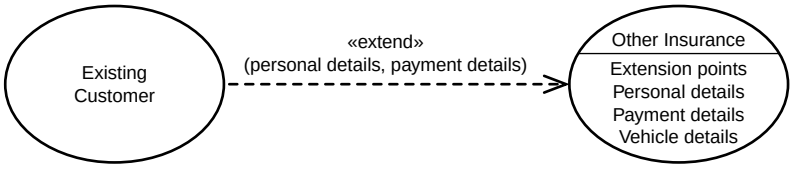
The `generalization` notation.



A more rigid notation than the generalization notation is the `<<extend>>` notation. This illustrates a situation where a Use Case has the same functionality as another Use Case, but it extends or overrides parts of that functionality. To show this, you must list the items within the base Use Case that the extending Use Case can extend. In addition, if the extending Use Case does extend some of these items (it is not mandatory to extend

every item on offer), you must place the names of these items in parentheses by the `<<extend>>` notation, as Figure A.4 illustrates.

FIGURE A.4
The `<<extend>>` notation.



Class Diagrams

Class diagrams show different objects within a system and how these objects associate with each other. In part, the diagram does this by showing the attributes and operations of each class within the system, and defining the constraints placed on these classes. The basic notation to show a class is a rectangular box containing the classname. However, you can add more information to the box, as you will see.

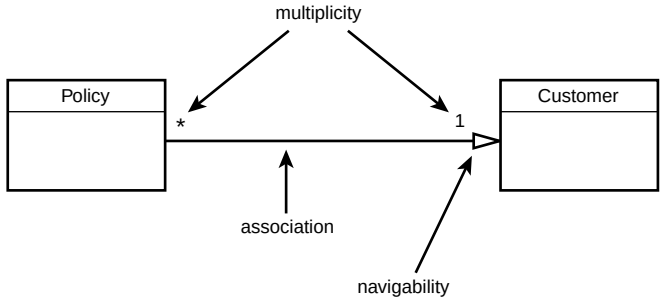
Note that the UML notation for class diagrams is extensive, so this appendix only shows you the most commonly used elements of the notation. Specifically, you will learn about five main types of notation:

- Associations
- Attributes
- Operations
- Generalization
- Constraints

Associations

If you want to show that an instance of a class has a relationship with an instance of another class, you join the two classes with a line, known as an *association*. Each end of the line is known as a *role*. You can further define the classes' relationships through the use of multiplicity and navigability. Multiplicity show how many objects can participate in a given relationship. For example, in Figure A.5, one customer (shown as 1) can have between zero and infinite policies (shown as *).

FIGURE A.5
An association.



There are four possible values for illustrating multiplicity:

- `1`—One instance
- `*`—Zero to infinite instances
- `0..1`—Zero or one instance
- `m..n`—User-defined between m and n

Navigability shows the responsibilities of classes. The arrow head points to the class that is not responsible in the relationship. For example, the Policy must say which Customer it belongs to in Figure A.5, but the Customer does not have to say which Policy instances it has. In this example, the responsibility is unidirectional, but in some situations, both parties might hold responsibilities. In these instances, you mark both roles with an arrow head; this is known as *bidirectional association*.

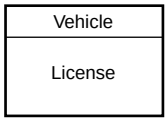
Attributes

Attributes have different meanings, depending on the level at which you are modelling. At a conceptual level, an attribute simply defines that a class has a certain feature, such as the `Vehicle` class having a license number. At the specification level, an attribute indicates that a `Vehicle` object has a way of setting the value of its license, and that it also can provide you with the value of that license. Finally, at the implementation level, an attribute indicates that a Vehicle has a field for the license.

Figure A.6 simply shows the name of the attributes, but it could show more detail. The UML defines the following, which you can use to define an attribute:

```
visibility name: type = defaultValue
```

FIGURE A.6
Attribute notation.



The visibility of the attribute can be either

- Public—Precede the name with a + (plus)
- Protected—Precede the name with a # (hash)
- Private—Precede the name with a – (minus)

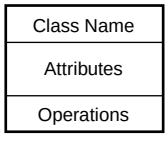
The `name`, `type`, and `default` values represent the attribute's name, type (for example, `String`), and default value (for example, `"unregistered"`). In some instances, you might want to show the multiplicity of an attribute, possibly to show whether the system requires it. To do this, place the multiplicity value (any one of those shown in the previous section) in square brackets after the attribute name, as shown in the following:

```
# licence[1]: String = "unregistered"
```

Operations

Operations are the processes that a class performs and, again, this can have different meanings at different levels. At the conceptual level, operations illustrate the responsibilities of a class. At a specification level, they should correspond to methods with public visibility. Finally, at the implementation level, they can correspond to methods with any degree of visibility. Figure A.7 illustrates that you show operations in the lower section of the box that represents the class.

FIGURE A.7
Operations notation.



You can provide quite a lot of information about the operation within the box. To do this, use whichever parts of the UML syntax you require:

```
visbility name(parameter list): return-type-expression { property-strings }
```

As you can see, the syntax is similar to that used to show an attribute, but with a few notable exceptions. The *`return-type-expression`* indicates a comma-separated list of return types—yes, the UML permits multiple return types. The *`parameter list`*, shown in parentheses, indicates a comma-separated list of parameters, whose syntax is as follows:

```
direction name: type = defaultValue
```

The different parts of the syntax have the same value as when you have previously encountered them, with the exception of direction. This shows whether a parameter is used for input, output, or both input and output. The values to indicate the direction are as follows:

- `in`—Parameter used only for input; this is the default value.
- `out`—Parameter used only for output.
- `inout`—Parameter used for both input and output.

Finally, to show how all this syntax ties together, the following example shows a protected operation that accepts two input parameters and returns a String object:

```
# myOperation(arg1: String, arg2: Integer=0):String
```

Generalization

Figure A.8 shows that a large, unfilled arrow head depicts generalization. Like an association, a generalization shows a relationship between two classes. However, unlike an association, it shows a special type of relationship where one class is the child of the other class. In other words, one class is a subclass or subtype of the other class. In practice, the meaning of generalization differs according to the context within which you construct the model. At the conceptual level, classes B and C are subtypes of A if all instances of them are also instances of A. At the specification level, you are dealing with interfaces. Thus, B and C are subtypes of A if they include all the elements of the interface of A. Finally, at the implementation level, you are showing inheritance between classes, where classes B and C inherit all the methods and fields of A; of course, they can override the methods they inherit.

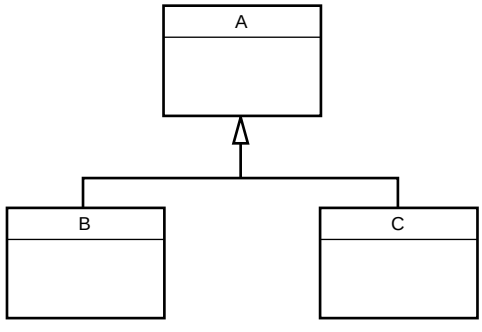


FIGURE A.8
Generalization notation.

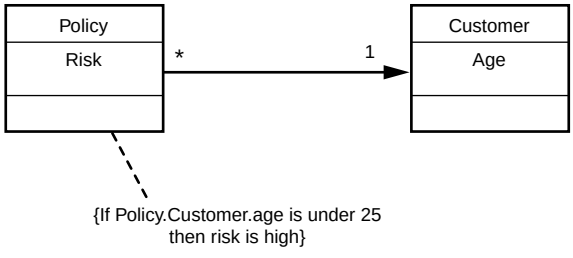
Constraints

So far in this appendix, you have looked at a number of ways to model the relationships between classes. Often, these relationships act to constrain the classes within that relationship. For example, multiplicity constrains a class in terms of how many instances of it can exist. The UML also provides further syntax that allows you to show constraints, which are otherwise concealed. The syntax is as follows:

```
{ description of constraint }
```

You connect the constraint description to a class by using a dashed line. The description of the constraint can take any form you want, but if you want to use a formal syntax, you can use the Object Constraint Language (OCL). The example that Figure A.9 shows simply uses plain text. As you can see, the model shows a constraint on the `Policy` class, which states that if a customer is under the age of 25 years old, the system should classify him or her as high risk.

FIGURE A.9
Using constraints notation.



Sequence Diagrams

Class diagrams show how classes interact with each other, but they do not give you a feel for how instances of those classes will interact in a real situation. Instead, sequence diagrams model this type of behavior. These diagrams do this by showing objects and the messages that pass between these objects. In this appendix, you will learn the UML syntax for sequence diagrams by exploring two examples, the first of which is shown in Figure A.10.

This first example shows most of the commonly used UML syntax for sequence diagrams. You place actors and boxes that represent objects or classes when their static methods are accessed at the top of the diagram. Descending from these items are dashed vertical lines that represent the life of the item. These lines are known as *lifelines*. The rectangles these lines pass through are *activations*, and these represent the time the objects require to do their work. The horizontal arrows represent *messages*—the communication between items that you model.

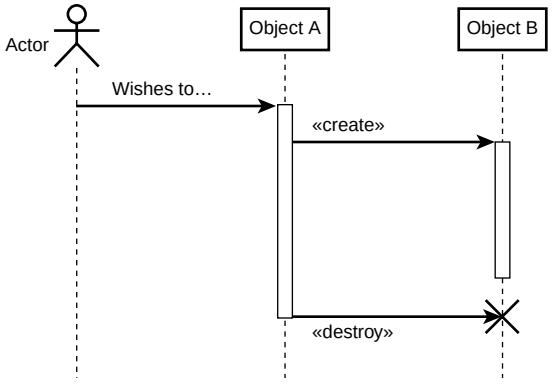


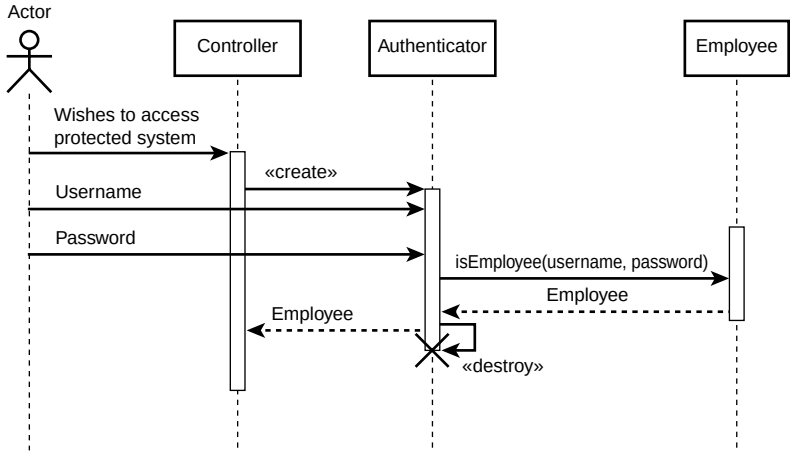
FIGURE A.10
Sequence Diagram #1.

As you can see, the actor sends a message to the first object, and the message in this instance take the form, "Wishes to..."—just a plain text description of the interaction. For example, this first object could be an instance of a servlet that the user calls. The first object then sends a message to the second object. In the diagram, the message is labeled `<<create>>`, which illustrates that the first object creates the second object. Likewise, the first object sends another message to the second object, but this time toward the end of its activation. This message is labeled `<<destroy>>`, and the lifeline of the second object is marked with a large X. This signifies that the first object deletes the second object. The other type of message that is used frequently, but is not shown in Figure A.10, is where the message label represents the name of the method an object calls on another object or class.

Figure A.11 shows a sequence diagram where a user wants to access some service of the system that requires authentication. The user sends a message to the `Controller` object, stating that he or she wants to access the system. This object creates an instance of the `Authenticator` class. The user then supplies this object with his or her username and password. The `Authenticator` receives these credentials and calls the static `isEmployee()` method of the `Employee` class. Notice how the message label in this instance is the method name together with a comma-separated parameter list. The `isEmployee()` method performs some form of processing on the parameters. In reality, this might involve looking up information in a data store. However, the example is simplified in this instance, so assume that this method does not require the services of any other class. You will notice that up to this point, all the UML syntax is the same as that shown in the previous example.

FIGURE A.11

Sequence Diagram #2.



After the `isEmployee()` method finishes its work, it returns an `Employee` object to the `Authenticator` object. As you can see, the UML syntax for this is a dashed horizontal line that points to the calling class or object and an optional label that indicates what is returned. At this point, the `Authenticator` object returns the `Employee` object to the `Controller` object. It then sends a `<<destroy>>` message to itself. This type of message is known as a *self-call* and is shown as a message arrow that leaves an activation and then doubles back to return to that activation. That's it; the sequence is complete.

The two previous examples show you the majority of the UML syntax you require to draw and understand sequence diagrams. The only other syntax that you may need to know relates to messages. Specifically, all the messages shown in this appendix have full arrow heads. This signifies that the messages are synchronous—the caller ceases operation until the called object returns. In reality, you might want to model situations where you require asynchronous messages. For example, you may have an object that creates new threads so that it can carry on with some other processing. To markup these types of messages, you show a half arrow head rather than a full arrow head.

APPENDIX B

SQL Reference

The Structured Query Language (SQL) is language that allows you to query, insert, and update data held in a relational database. The language has evolved during the past three decades to the latest incarnation of the language, which is called SQL99—otherwise known as SQL3. SQL99 is a standard published by the American National Standards Institute (ANSI) and ratified by the International Standards Organization (ISO). Today, SQL is the eminent language used by developers to work with relational databases.

> Unlike some standards, SQL99 is not available on a "free access" model. You can purchase sections of the standard, at $18 per section from `http://www.ncits.org/`.

Each provider of a Relational Database Management System (RDBMS) provides their own implementation of the language, but these implementations generally still comply with the standard. For example, the SQL statements, clauses, and functions you use with Oracle differ from those you use with MySQL, but both conform to the SQL standard. The good news is that you will

find that each vendor's implementation conforms to the standard to such an extent that you do not have to learn a complete new set of commands to work with their product. However, you will have to refer to their documentation to discover which parts of SQL99 they adhere to and what vendor-specific extensions they have added to their product. In the process of using this book, you may have used one of any number of RDBMSs. So, rather than provide a reference to a specific vendor's implementation of the language, this appendix provides a reference to the most commonly used statements and clauses of the SQL99 standard. You should find that this reference, irrespective of the RDBMS you use, is generic enough to allow you to build the most commonly used SQL statements and execute them against your RDBMS with only a small number of changes to your code.

Commonly Used SQL Statements (SQL99)

The following provides a reference to the SQL statements that you might use on a day-to-day basis. It does not intend to provide a complete reference, and it also does not aim to illustrate vendor implementation-specific syntax. For a fuller reference or an exact guide to your chosen RDBMS, please refer to your vendor's documentation.

ALTER TABLE

```
ALTER TABLE table_name {
    ADD [COLUMN] column_name datatype attributes
    | ALTER COLUMN column_name SET DEFAULT default_value
    | ALTER COLUMN column_name DROP DEFAULT
    | ALTER COLUMN column_name ADD SCOPE table_name
    | ALTER COLUMN column_name DROP SCOPE [RESTRICT | CASCADE]
    | DROP COLUMN column_name [RESTRICT | CASCADE]
    | ADD table_constraint_name
    | DROP CONSTRAINT table_constraint_name [RESTRICT | CASCADE]
}
```

Description:

The `ALTER TABLE` statement alters the columns and constraints of a table. As you can see, this statement provides a number of ways to add, modify, or drop a column from a table. Statements that utilize `RESTRICT` and `CASCADE` allow you to dictate how the database should behave if either dropping or modifying a column impacts on foreign or primary keys. Specifically, `RESTRICT` forces the database to throw an exception, or simply abort the command, if the database contains foreign keys that reference a primary key that you want to drop or modify. `CASCADE` instructs the database to drop any foreign keys that reference a primary key that you want to drop or modify.

The constraints options allow you to place table-wide restrictions on the type of data a user can enter into a table's rows. In the context of constraints, RESTRICT and CASCADE have the same meaning as with columns.

CREATE TABLE

```
CREATE [GLOBAL TEMPORARY | LOCAL TEMPORARY] TABLE table_name
    [ON COMMIT [PRESERVE ROWS | DELETE ROWS] ] {
        column_name datatype [(length)] [NULL | NOT NULL],...n
        | [LIKE table_name]
        | [table_constraint][,...n] ]
}
```

Description:

The CREATE TABLE statement creates a new table. Both the TEMPORARY options are optional, and they both create temporary tables that the database discards at the end of a session. The GLOBAL option creates a table that is available to all user sessions; whereas, the LOCAL option creates a table that is only available to the user session that creates it. The ON COMMIT clause is used only with temporary tables, and it has two options. The first preserves any modifications to a temporary table when you issue a COMMIT statement. The second, DELETE ROWS, deletes the rows of the temporary table when you issue a COMMIT statement; this is the default behavior that SQL99 defines.

The LIKE option allows you to create a new table based on the definition of an existing table. For example, you create a new table that has the same column names as an existing table.

The optional table constraints clause allows you to place constraints, or tests, on the data types that a user can insert into a column. For example, you could stipulate that a column is the primary key for the table or that it must contain a unique value.

CREATE VIEW

```
CREATE VIEW view_name [(column [,...])] AS SELECT_statement
[WITH [CASCADE | LOCAL] CHECK OPTION]
```

Description:

The CREATE VIEW statement creates a new view based on a query. The list of columns is optional. It allows you to assign names to each of the columns of the view. If you do not provide a list of columns, the statement uses the column names returned by the SELECT statement.

The optional WITH CHECK OPTION option ensures that a view can only insert, modify, or delete data that it can read from the view's base table. The LOCAL option only applies the

check to the current view, whereas the CASCADE option applies the check to the current view and all the views on which it is built.

DELETE

```
DELETE FROM [owner.]table_name [WHERE clause]
```

Description:

The DELETE statement deletes rows from a table. If you use the statement without the optional WHERE clause, it deletes all the rows in the table. For an explanation of the WHERE clause, see the "Commonly Used SQL Clauses" section later in this appendix.

DROP TABLE

```
DROP TABLE table_name [RESTRICT | CASCADE]
```

Description:

The DROP TABLE statement drops a table—including constraints, indexes, and triggers—from a database. RESTRICT forces the database to throw an exception or abort the operation if any views or constraints reference the table. In contrast, CASCADE drops all views or constraints that reference the table.

DROP VIEW

```
DROP VIEW view_name [RESTRICT | CASCADE]
```

Description:

The DROP VIEW statement drops a view. RESTRICT forces the database to throw an exception or abort the operation if any views or constraints reference the table. In contrast, CASCADE drops all views and constraints that reference the table.

INSERT

```
INSERT INTO [database_name.]owner. (table_name | view_name) [(column [,...])]
{[DEFAULT] VALUES | VALUES (value[,...]) | SELECT_statement }
```

Description:

The INSERT statement inserts new rows of data into either a table or view.

The list of columns is optional, but if you omit it, the statement will either assume the order of the columns and insert data on this assumption (inserting either null or default values into the remaining columns, or the RDMBS will throw an exception, for example, MySQL throws a Column Count Does Not Match Value Count exception. The

statement accepts data in three ways. The first is using the `DEFAULT VALUES` method, which inserts the database implementation's default values, if the implementation supports default values (most do), into the table. The second, `VALUES`, contains a list of comma-separated values that insert into the specified columns. Finally, you can use a `SELECT` statement, which is described next in this appendix.

SELECT

```
SELECT [ALL | DISTINCT] [select[,...]] FROM table_name [,...]
[JOIN join_condition]
[WHERE search_condition]
[GROUP BY group_by_expression[,...]]
[HAVING search_condition [,...]]
[ORDER BY order_expression [ASC | DESC] ]
```

Description:

The `SELECT` statement retrieves data from a table. The basic form of the statement is shown in the first line of the syntax. The list of select items can consist of column names, expressions, local and global variables (where supported), mathematical calculations, or a wildcard `*` (returns all columns). In addition, you can specify an alias for the returned columns. For example, to select the column `first_name` but reference it as `fore_name`, you use the syntax `SELECT first_name AS fore_name`. You can also apply this syntax to a table name by using the `FROM` clause. The default behavior of the statement is to return all records, including duplicates and those containing only default values. If you use the `DISTINCT` option, the statement does not return duplicate records.

> The final section of this appendix, "Commonly Used SQL Commands," describes the use of the `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses.

The `JOIN` operation allows you to modify a `SELECT` statement so that it returns a result set based on a relationship you define (a join) between columns in two different tables. There are five main types of join.

Inner Join

The Inner Join, which is also known as an EquiJoin, joins two tables where a Boolean expression you specify returns `true`. For example, the following `JOIN` operation joins the `products` table and `suppliers` table by their `supplier_id` columns. In this instance, the statement will only return the records where supplier IDs correspond in both tables:

Although the SQL99 standard syntax for an inner join is `JOIN`, many RDBMS use the syntax `INNER JOIN`.

```
SELECT * FROM products JOIN suppliers
ON products.supplier_id=suppliers.supplier_id
```

CROSS JOIN

The `CROSS JOIN` joins two tables by Cartesian Product—each record in the first table joins to all the records in the second table. Because a `CROSS JOIN` joins every record in two tables, you do not specify columns to join. Thus, the syntax for this join appears as follows:

```
SELECT * FROM products CROSS JOIN suppliers
```

Using a `CROSS JOIN` on a live database can result in an enormous result set. You can very easily place an excessive load on a system using this command, and this can adversely affect the normal operation of your database server.

Left Outer Join

The `LEFT JOIN` joins two tables and returns all the values from the left table and only the matching records from the right table. Where no match occurs, a `NULL` value is returned as the value of the right table. For example, the following statement returns all records that have supplier IDs in the `products` table, but only those records from the `suppliers` table that have a corresponding supplier ID in the `products` table:

```
SELECT * FROM products LEFT JOIN suppliers
ON products.supplier_id=suppliers.supplier_id
```

Right Outer Join

The `RIGHT JOIN` joins two tables and returns all the values from the right table and only the matching records from the left table. Where no match occurs, a `NULL` value is returned as the value of the left table. For example, the following statement returns all the supplier IDs from the `suppliers` table, but only those from the `products` table that have a corresponding entry in the `suppliers` table:

```
SELECT * FROM products RIGHT JOIN suppliers
ON products.supplier_id=suppliers.supplier_id
```

FULL JOIN

The FULL JOIN joins two tables, and returns all the rows from both tables. For example, the following statement returns all the supplier IDs from both tables regardless of whether they match:

```
SELECT * FROM products FULL JOIN suppliers
ON products.supplier_id=suppliers.supplier_id
```


UPDATE

```
UPDATE {table_name | view_name}
SET {column_name | variable_name} = {DEFAULT | expression} [,...n]
WHERE conditions
```

Description:

The UPDATE statement changes data in a table or view. The statement centers on the SET clause that allows you to set a named column to a new data value. You can express the data value either as DEFAULT—the column's default value —or as an expression. The expression can be an expression, such as age+10; or a value, such as 'New Value'. The WHERE clause is optional, but if you omit it, the statement updates all the records in the column.

Commonly Used SQL Clauses

The following list describes the SQL clauses that you will use most frequently. In relation to the previous SQL statements, the WHERE clause is used with the DELETE, UPDATE, and SELECT statements; the remaining clauses are only used with the SELECT statement.

FROM

```
FROM table_name [,...]
```

Description:

The FROM clause specifies the tables from which a statement should retrieve data. You can combine the FROM clause with AS to assign aliases to table names. For example, to assign an alias of products to a table called us_products, you use the following syntax:

```
FROM us_products AS products.
```

WHERE

```
WHERE Boolean_expression
```

Description:

The WHERE clause restricts a statement so that it only returns those rows or records for which a Boolean expression that you define evaluates as true. For example, to delete only those rows from a table where a record has an employee name of John Smith, you execute the following:

```
DELETE FROM employee_table WHERE employee_name = 'John Smith'
```

GROUP BY

```
GROUP BY column_name [,...]
```

Description:

The GROUP BY clause divides the result of a SELECT statement into logical groups. The clause allocates data to these groups by grouping data that has identical values for the column names you specify. Typically, you do not use this clause to group the data that is displayed (use the ORDER BY clause to achieve this), but instead you use it together with an aggregate function, such as MAX, AVG, or COUNT. For example, to get a breakdown of how many employees live in each state, you issue the following command:

```
SELECT COUNT(*) FROM employee_table GROUP BY home_state
```

HAVING

```
HAVING search_condition
```

Description:

The HAVING clause works with the GROUP BY clause in much the same way as the WHERE clause works with the SELECT statement. It allows a statement to return only items within groups that conform to a condition the HAVING clause defines. For example, the following command lists the average employee salary for all departments where the average salary is greater than 30,000 dollars:

```
SELECT department, AVG(salary)
FROM employee_table
GROUP BY department
HAVING AVG(salary)>30000
```

ORDER BY

```
ORDER BY { (column_name | column_position) [ASC | DESC]] }[,...]
```

Description:

The `ORDER BY` clause defines the order in which a statement returns a result set. The `ASC` and `DESC` options dictate whether the statement should return the results in ascending or descending order, respectively; ascending order is the default. The `column_name` is the name of the column by which to sort. The column position allows you to stipulate an ordinal position that relates to the column names or aliases you specify when creating the statement. For example, the following statement returns all the employee forenames and surnames from an `employee` table and orders the results by surname in descending order:


```
SELECT forename, surname
FROM employee_table
ORDER BY 2,1
```

APPENDIX C

An Overview of XML

Throughout the book, there are many examples of how XML can be used within J2EE applications. Day 16, "Integrating XML with J2EE," and Day 17, "Transforming XML Documents" cover the topic of XML in some detail. This appendix is here to provide a quick reference that should enable you to comprehend XML examples used in this book.

XML is often described as portable data that co-exists alongside Java's portable code. Many new initiatives in Java use XML in a central role, so it is rapidly becoming a standard part of the Java developer's toolkit.

This appendix examines:

- The syntax and structure of an XML document
- Ways of defining XML structure, such as DTD and XML schema
- How different XML dialects can be identified

What Is XML?

XML has arisen from the need for a portable data format.

Essentially, XML is a standard for representing data in a text document. XML provides a framework for representing almost any kind of data, which is one of the reasons why it has attracted so much interest.

An XML document consists of text-based tags used to provide the document structure (similar to those used in HTML) together with the data itself. All XML documents consist of elements and optional declarations and comments.

Elements

An element has the following form:

```
<start_tag attributes>body<end_tag>
```

For example,

```
<book title="J2EE in 21 Days">A very useful book</book>
```

In XML, unlike HTML, the tags are not predefined. As the author of an XML document, you are free to invent whatever tags are appropriate for the data you are describing.

When defining an XML tag, you may include attributes that further describe the tag. In the previous example, the title of the book is supplied as an attribute to the book tag.

The body of an element is all the text, including any nested tags, enclosed by the start and end tags.

An element need not have any attributes or even any body.

Tag names must start with a letter or underscore and can contain any number of letters, numbers, hyphens, periods, or underscores, but they cannot include spaces.

All XML is case sensitive, and attributes must be quoted (both single and double quotes are accepted). The following are alternative forms for an element:

```
<tag>text</tag>
<tag attribute="text">text</tag>
<tag attribute="text"></tag>
<tag></tag>
<tag attribute="text"/>
<tag/>
```

The last two in this list show examples where the start and end have been combined. This is done simply to reduce clutter in the document.

Tags must nest. That is, an end tag must close the textually preceding start tag. For example,

```
<B><I>bold and italic</I></B>
```

The following is not well-formed XML:

```
<B><I>bold and italic</B></I>
```

To be well-formed XML, the `</I>` end tag must precede the `</B>` so the tags nest correctly.

The tags provide

- Information about the meaning of the data
- The relationships between different parts of the data

There must be exactly one top level element in an XML document, called the root element, which must enclose all the other elements in the document.

The following is a well-formed XML document:

```
<jobSummary>
  <job customer="winston" reference="Cigar Trimmer">
    <location>London</location>
    <description>Must like to talk and smoke</description>
    <skill>Cigar maker</skill>
    <skill>Critic</skill>
  </job>
</jobSummary>
```

The root element is `<jobSummary...</jobSummary>`. The `<job>` element has two attributes and enclosed elements.

Declarations

Declarations are used to provide information to the XML parser. They are of two forms. The first is a Processing Instruction and is enclosed in <? ... ?>.

The following example tells the parser that the document has been written using XML version 1.0 and the UTF-8 character encoding:

```
<?xml version ="1.0" encoding="UTF-8"?>
```

The second form of declaration is an XML Document Type Declaration and is preceded enclosed in `<! ... >`.

Do not confuse a Document Type Declaration, which is the XML element containing declarations indicating the grammar that should be applied to validate an XML document, with the grammar itself, which is called a Document Type Definition (DTD). DTDs are explained later in this appendix.

```
<!DOCTYPE jobSummary SYSTEM "jobSummary.dtd">
```

Document Type Declarations are used to inform the parser of the correct structure of the XML document and to validate the XML. There is more information on the different type of document type declarations in section "Document Type Definition" later in this appendix.

If declarations appear in an XML document, they must precede the root element. This section is usually referred to as the *prolog*.

Comments

As well as elements and declarations, an XML document can contain comments that help to clarify the document content for human readers. Comments can be used anywhere within an XML document that a tag could appear. An example is as follows:

```
<!-- This is a really good book -->
```

Special Characters

The characters in Table C.1 have a special meaning in XML and, if required in the contents of an element, they must be replaced with the symbolic form.

TABLE C.1 Special XML Characters

Character	*Name*	*Symbolic Form*
`&`	(ampersand)	`&`
`<`	(open angle bracket)	`<`
`>`	(close angle bracket)	`>`
`'`	(single quotes)	`'`
`"`	(double quotes)	`"`

Other special characters, such as non-printing characters, that may cause problems during processing, should be replaced by entities that give their decimal value. For example, `^A` becomes `&#01;`.

If you are familiar with HTML, you will recognize the technique of replacing certain characters or including characters not found in standard character sets (such as ©) with a character entity that is either `&name;` or `&#nnn` (where *nnn* is a numeric representing the character). As an HTML user, you are also probably aware that browsers can interpret character entities differently. This means the character encoding you are familiar with may not conform to the standard. Refer to the W3C Web site to find a list of the character entities for the ISO-8859-1 (Unicode 2.0) character set. Only those character entities defined in the standard should be used in XML.

For data containing large amounts of special characters, you can use a `CDATA` section. This begins with the string `<![CDATA[` and ends with `]]>`. Any characters between the start and end of a `CDATA` section are not processed by the parser and are just treated as a text string.

Namespaces

Namespaces are used to scope tags within a document. The use of multiple namespaces allows different tags to have the same name but different meanings in a single XML document.

An attribute called `xmlns` (XML Name Space) is added to an element tag in a document and is used to define a namespace for the body of the element.

The following is a document with two namespaces:

```
<?xml version ="1.0"?>
<jobSummary xmlns:ad="ADAgency" xmlns:be="BEAgency">
  <ad:job customer="winston" reference="Cigar Trimmer">
    <ad:location>London</ad:location>
    <ad:description>Must like to talk and smoke</ad:description>
    <ad:skill>Cigar maker</ad:skill>
    <ad:skill>Critic</ad:skill>
  </ad:job>
  <be:job>
    a completely different form of the job element
  </be:job>
</jobSummary>
```

Enforcing XML Document Structure

If XML is used to transfer information between applications, there needs to be a mechanism for ensuring that the XML is not only syntactically correct but also is structurally correct. In fact, there are two common mechanisms for this:

- Document Type Definitions
- XML Schemas

Document Type Definition (DTD)

A Document Type Definition (DTD) is a way of defining the structure of an XML document. DTD elements can be included in the XML document itself or in a separate external document. The syntax used to define a DTD is different from XML itself.

The following is an example DTD that describes the jobSummary XML:

```
<!DOCTYPE jobSummary>
<!ELEMENT jobSummary (job*)>
<!ELEMENT job (location, description?, skill*)>
<!ATTLIST job customer CDATA #REQUIRED>
<!ATTLIST job reference CDATA #REQUIRED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT skill (#PCDATA)>
```

The `!DOCTYPE` element must include the name of the root element. If the remainder of the document type definitions are stored in an external file, it will have the following form:

```
<!DOCTYPE root_element SYSTEM "external_filename">>
```

If the definitions are included in the XML document itself, the `!DOCTYPE` element must appear in the document prolog before the actual document data begins. In this case, the `!DOCTYPE` element must include all the DTD elements with the following syntax:

```
<!DOCTYPE jobSummary [
<!ELEMENT jobSummary (job*)>
<!ELEMENT job (location, description?, skill*)>
<!ATTLIST job customer CDATA #REQUIRED>
<!ATTLIST job reference CDATA #REQUIRED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT skill (#PCDATA)>
]>
```

The other elements (`!ELEMENT` and `!ATTLIST`) are described in this section.

Elements

Element declarations take the following form:

```
<!ELEMENT element_name (content)>
```

where *`element_name`* is the XML tag and *`content`* is one or more of the values shown in Table C.2.

TABLE C.2 DTD Content Specifications for Elements

Content Type	*Syntax*	*Element contains*
Element	`<!ELEMENT tag (sub1)>`	Sub-element only
`#PCDATA`	`<!ELEMENT tag (#PCDATA)>`	Text only
`EMPTY`	`<!ELEMENT tag (EMPTY)>`	Nothing
`ANY`	`<!ELEMENT tag (ANY)>`	anything (text or elements)

`#PCDATA` limits the content of the element to character data only; nested elements are not allowed. Do no confuse with CDATA sections in XML that are used to present large areas of un-interpreted text.

The characters in Table C.3 can be used to combine multiple element content types to define more complex elements.

TABLE C.3 Content Characters Used in DTD Definitions

Character	*Meaning*
,	Sequence operator, separates a list of required elements
*	Zero or more (not required)
+	One or more (at least one required)
?	Element is optional
|	Alternate elements
()	Group of elements

The following is a declaration for the `job` element:

```
<!ELEMENT job (location, description?, skill*)>
```

The `job` element consists of, in order, one `location`, an optional `description`, and an optional list of `skill` elements.

Attributes

Attribute declarations take the following form:

```
<!ATTLIST element_name attribute_1_name (type) default-value
                       attribute_2_name (type) default-value>
```

An attribute type can be any one of the types shown in Table C.4, though CDATA (text) is the most common.

TABLE C.4 DTD Attribute Types

Type	*Attribute is a…*
CDATA	Character string.
NMTOKEN	Valid XML name.
NMTOKENS	Multiple XML names.
ID	Unique identifier.
IDREF	An element found elsewhere in the document. The value for IDREF must match the ID of another element.
ENTITY	External binary data file (such as a GIF image).
ENTITIES	Multiple external binary files.
NOTATION	Helper program.

The default-value item can also be used to specify that the attribute is #REQUIRED, #FIXED, or #IMPLIED. The meanings of these values are presented in Table C.5.

TABLE C.5 DTD Attribute Default Values

Default Value	*Meaning*
#REQUIRED	Attribute must be provided.
#FIXED	Effectively a constant declaration. The attribute must be set to the given value or the XML is not valid.
#IMPLIED	The attribute is optional and the processing application is allowed to use any appropriate value if required.

Entity References

Another DTD element not mentioned so far is an entity reference. An entity reference has more than one form. The first, called a general entity reference, provides shorthand for often-used text. An entity reference has the following format:

```
<!ENTITY name "replacement text">
```

This is, in fact, how the special characters are handled. The character entity & is defined as <!ENTITY & "&">.

The entity reference called `name` can be referred to in the XML document using `&name;`, as shown in the following:

```
<!DOCTYPE book [
...
<ENTITY copyright "Copyright 2002 by Sams Publishing>
]>

<book title="J2EE in 21 Days">A very useful book &copyright;</book>
```

The second form, called an external entity reference, provides a mechanism to include data from external sources into the document's contents. This has the following format:

```
<!ENTITY name SYSTEM "URI">
```

For example, if the file `Copy.xml` that can be retrieved from the Sams Web site contains the following XML fragment

```
<copyright>
  <date>2002</date>
  <publisher>Sams Publishing</publisher>
</copyright>
```

this can be referenced in any XML document as follows:

```
<!DOCTYPE [
...
<ENITITY copyright http://www.samspublishing.com/xml/Copy.xml>
]>
<book>
  <title>J2EE in 21 Days>
..&copyright;
  <synopsis>All you need to know about J2EE</synopsis>
</book>
```

XML Schema

Like DTDs, an XML Schema can be used to specify the structure of an XML document. In addition, it has many advantages over DTDs:

- Schemas have a way of defining data types, including a set of pre-defined types.
- A schema is namespace aware.
- It is possible to precisely specify the number of occurrences of an element (as opposed to a DTD's imprecise use of `?`, `*`, and `+`) with the `minOccurs` and `maxOccurs` attributes.
- The ability to restrict the values that can be assigned to predefined types.
- A schema is written in XML.

The following is a schema to define the `jobSummary` XML:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
➥ elementFormDefault="qualified">

  <xsd:element name="jobSummary">
    <xsd:complexType>
      <xsd:sequence>
<xsd:element name="job" type="jobType" minOccurs="0"
➥ maxOccurs="unbounded"/>
      </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

  <xsd:complexType name="jobType">
    <xsd:sequence>
      <xsd:element name="location" type="xsd:string"/>
       <xsd:element name="description" type="xsd:string"/>
<xsd:element name="skill" type="xsd:string" minOccurs="1"
➥ maxOccurs="unbounded"/>
    </xsd:sequence>
      <xsd:attribute name="customer" type="xsd:string" use="required"/>
      <xsd:attribute name="reference" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

In schemas, elements can have a `type` attribute that can be one of the following:

- `string` Any combination of characters
- `integer` An integral number
- `float` A floating-point number
- `boolean` `true`/`false` or `1`/`0`
- `date` yyyy-mm-dd

There are considerably more predefined simple data types. A full list can be obtained from the W3C Web site.

Or an element can be a complex type, which is a combination of elements or elements and text.

The number of times an element can appear is controlled by two attributes:

- `minOccurs`
- `maxOccurs`

For example, the following `skill` element must appear at least once and can occur any number of times.

```
<xsd:element name="skill" type="xsd:string" minOccurs="1"
➥ maxOccurs="unbounded"/>
```

Elements can be made optional by setting the value of the `minOccurs` attribute to `0`.

Element attributes can be declared with a `use` attribute to indicate whether the element attribute is `required`, `optional`, or even `prohibited`.

A declaration of a complex type generally includes one of the following that specifies how the elements appear in the document:

- all—All the named elements must appear, however they may be in any order.
- choice—One, and only one, of the elements listed must appear.
- sequence—All the named elements must appear in the sequence listed.

Where to Find More Information

More information on XML standards can be found at various Web sites, the most important being the W3C Web site, which is found at `http://www.w3.org`.

Day 16, "Integrating XML with J2EE," covers in more detail the subject of creating and validating XML. It introduces the Java API for XML Processing (JAXP) that allows you to use J2EE to parse and create XML.

Other related XML subjects, such as XSLT, Xpath, and XPointer, are covered on Day 17, "Transforming XML Documents." A brief introduction to these subjects is given in this section.

XSL is a stylesheet language for XML. XSL specifies the styling of an XML document by using XSL Transformations to describe how the document is transformed into another XML document.

XSLT is a language for transforming XML documents into other XML documents. A transformation expressed in XSLT is called a stylesheet.

XPointer provides a mechanism to "point" to particular information in an XML document.

XPath is a language for identifying parts of an XML document; it has been designed to be used by both XSLT and XPointer. XPath gets its name from its use of a compact *path* notation for navigating through the hierarchical structure of an XML document.

With the XPath notation, it is, for example, possible to refer to the third element in the fifth `Job` node in a XML document.

XPath is also designed so that it can be used for matching (testing whether or not a node matches a pattern). The form of XPath used in XSLT.

Everything in this appendix and a lot more is also covered in some detail in the *Sams Teach Yourself XML in 21 Days*, Shepherd, ISBN 0-672-32093-2. This book covers everything you need to know about XML to "hit the ground running."

APPENDIX D

The Java Community Process

Many of the lessons in this book have referred to JSRs (Java Specification Request). If you are not familiar with the Java Community Process (JCP), you may wonder exactly what a JSR is and how it affects J2EE technologies. This appendix provides you with an introduction to both JSRs and the JCP, and explains why they affect J2EE and you as a developer.

Introducing the JCP

The Java platform is developed within an open framework, unlike some other technologies. The JCP is the framework within which this open development occurs. It involves a number of interested parties, potentially including yourself, who develop or modify:

- Java technology specifications
- Technology Compatibility Kits (TCK)
- Reference Implementations (RI)

The JCP revolves around JSRs, which are the formalized requests that JCP members make when they want to either develop a new Java technology specification or modify an existing specification. Before you discover what is involved in the process of converting a JSR to a finalized specification, you will learn who is involved in the JCP.

Getting Involved

There are five main groups involved with the JCP. Each group plays a defined role that ensures that the JCP delivers Java technology specifications that meet the needs of developers and organizations, and ensure the continued stability and cross-platform compatibility of Java technologies.

JCP Members

Any individual or organization can become a member of the JCP. To become a member, you must sign the Java Specification Agreement (JSPA) and pay a fee, which, at the time of writing, is $5000/ann for commercial entities and $2000/ann for all other entities.

JCP members are responsible for the submission of JSRs that are then further developed by Expert Groups. These groups consist of experts that JCP members may nominate either themselves or other members for. One JCP member will lead each Expert Group and is responsible for forming the group and adding experts to that group. JCP members also have the right to vote on Executive Committee ballots; you will learn about these a little later.

Expert Groups

Each expert group is responsible for forming a specification and its RI and TCK from a JSR. In addition, once they form the specification, they are responsible for the maintenance of that specification.

When JCP members make nominations for Expert Group members, they ensure that the group will consist of individuals who are experts in the technology to which the specification relates. In addition, they ensure that the Expert Group includes enough depth and breadth of knowledge to enable the final specification to be of real use to developers and organizations.

The Public

Any member of the public can become involved with the JCP without having to become a full member of the JCP or pay a fee. The main ways that members of the public can become involved are by reviewing and commenting on

- Any specification JCP members develop
- Any new or revised JSR
- Proposed error corrections and modifications to existing specifications

Process Management Office (PMO)

The PMO is a group within Sun Microsystems that manages the day-to-day running of the JCP. The group does not involve itself with actual formation of JSRs and the final specifications.

Executive Committees

There are two Executive Committees, each overseeing different elements of the Java platform, namely the Standard Edition, Enterprise Edition, and Micro Edition. It is the responsibility of an Executive Committee to oversee the work of the Expert Groups to ensure that specifications do not overlap or conflict with each other. The Executive Committee is not involved with the JCP on a day-to-day process, but, instead, reviews the work of Expert Groups at defined points of the JCP. Specifically, an Executive Committee selects JSRs for development, provides guidance for the PMO, and approves

- Draft specifications
- Final specifications
- Maintenance revisions of a specification
- The transfer of maintenance responsibilities between JCP members

Each Executive Committee consists of sixteen seats. Of these, only one is permanent—held by Sun Microsystems. Of the remaining seats, ten are ratified and five are elected. Each of these seats is held for three years, and its holder is determined on a rolling basis; thus, five seats are either ratified or held open for election each year.

Understanding the JSR Process

There are several stages to transforming an initial JSR to a final specification, and each involves different entities concerned with the JCP. However, the process consists of three main stages, which are shown by Figure D.1:

FIGURE D.1
The JCP process.

As you can see, the process consists of three main sections. The first, Initiation, is where a JCP member submits a JSR. This JSR is open for review by JCP members and the public. Once reviewed, the Executive Committee decides whether to approve the request. If the request is approved, the process moves into the Community Draft stage.

This stage is where the Expert Group is formed. The Expert Group then writes the first draft of the specification and makes the draft available for Community Review by JCP members. The Expert Group may update the draft at this point, or they may pass the draft immediately to the Executive Committee for approval. If the draft is approved, the process moves into the final stage—Public Review and Specification Finalization.

This final stage commences with the group posting the draft on the Internet so that the public can review and comment on it. After the public review is complete, the Expert Group may modify the draft to include feedback from the public. At this point, the group prepares the proposed final draft and ensures that the RI and TCK are complete. After the proposed final draft is complete, the group passes it to the Executive Committee for final approval. If the specification is approved, it is then released.

After the Expert Group release the final specification, the specification is not simply abandoned. Instead, it is subject to an ongoing process of review and modification. Within this process, there might be requests for revisions or enhancements to the specification, or a need for further clarification and interpretation of the specification. The Executive Committee is responsible for reviewing each proposed change, and they will decide on a suitable course of action to implement the change.

Taking the Next Step

This appendix has provided you with a brief overview of the Java Community Process, the roles you can play within in it, and the lifecycle of a Java Specification Request. If you want to find out more or get involved, you can do so by visiting `http://www.jcp.org`.

If you simply want to browse the JSR archive, you can do so by visiting `http://www.jcp.org/jsr/overview/index.jsp`.

GLOSSARY

This appendix lists new or uncommon terms used throughout this book and provides a brief definition.

3-Tier The 3-tier model for enterprise applications, as used by J2EE, that splits the application functionality into three parts: presentation, business, and integration. Components that deliver these three types of functionality will typically live on their own tier of servers, so that the three types of functionality are physically as well as logically separated. See also n-tier.

Active Directory Active Directory is Microsoft's directory service, first delivered as part of Microsoft Windows 2000.

ANSI The American National Standards Institute is a private, non-profit organization that administers and coordinates the U.S. voluntary standardization and conformity assessment system.

ANSI SQL ANSI SQL represents a standard for SQL programming that is independent of any one specific implementation. The first ANSI SQL standard was published in 1989, but most vendors now support the update published in 1992. See also ANSI SQL 92.

ANSI SQL 92 ANSI SQL 92 refers to the version of the SQL specification published by ANSI in 1992. This forms the basis of most current SQL implementations by major vendors. See also SQL.

Apache Software Foundation (or just Apache) The Apache Software Foundation is an umbrella organization that supports a range of open-source projects being pursued under the Apache banner. Notable among these projects are the Jakarta Project, which delivers the Tomcat servlet and JSP implementation, together with the XML Project, which oversees the development of the Crimson and Xerces parsers, the Xalan XSLT processor and the Axis SOAP engine. See also Axis, Jakarta.

Application Client A J2EE application client is a client-side J2EE component that has access to a subset of the J2EE APIs provided by the J2EE client container. A J2EE application client must be invoked within the context of a J2EE client container, such as the `runclient` container provided by the J2EE RI.

Application Layer The application layer is a term used to refer to the logical layer containing the interaction with the user of the application. This can include not only Web-based interaction using servlets and JSPs with a Web browser, but also application clients.

Application Server An application server is a server-side container for components of an n-tier application. In Java terms, a typical application server will provide all of the J2EE APIs and container types. Application servers can also provide additional functionality, such as CORBA, COM, or Web Service support. Common application servers include BEA WebLogic, IBM WebSphere, iPlanet Application Server (iAS), and JBoss.

Auxiliary Deployment Descriptor The deployment descriptors provided with J2EE components and applications provide standard information about the properties and configuration of those components and applications. The auxiliary deployment descriptor defines additional, non-standard information about the J2EE application or component that is used by a specific J2EE container or application server. Hence, the contents of the auxiliary deployment descriptor are specific to that environment. See also Deployment Descriptor.

Axis The Apache Axis project is part of the Apache XML project. Axis is a Java-based SOAP toolkit that allows you to build and invoke Web Services from Java components. See also Apache.

Bean See JavaBean.

Bean-Managed Persistence (BMP) See BMP.

Bean-Managed Transaction Demarcation (BMTD) See BMTD.

BMP (Bean-Managed Persistence) An Entity EJB can take responsibility for persisting and retrieving its own internal state when prompted by its container. This is commonly done by including JDBC code in the appropriate lifecycle methods. This style of Entity EJB persistence is termed Bean-Managed Persistence. See also CMP.

BMTD (Bean-Managed Transaction Demarcation) An EJB can take control of its own transactions by making API calls to start and end transactions. This is termed Bean-Managed Transaction Demarcation. See also CMTD.

Business Tier The set of machines on which the business components execute in an n-tier or 3-tier application. See also n-tier, 3-tier.

Client Tier See Presentation Tier.

CMP (Container-Managed Persistence) An Entity EJB can delegate responsibility for persisting and retrieving its internal state to its container. This is termed Container-Managed Persistence. See also BMP.

CMR (Container-Managed Relationships) Under EJB 2.0 (and J2EE 1.3), it is possible to specify relationships between Entity EJBs in such a way that the container will automatically manage the lifecycle of the whole interconnected web of entities according to those relationships. This means that entities that are referenced by other entities will automatically be instantiated and populated when required, with no need for code in either the client or the containing Entity. Such relationships are termed Container-Managed Relationships.

CMTD (Container Managed Transaction Demarcation) An EJB can delegate control of its transactions to its container, which will start and end transactions on behalf of the EJB. This is termed Container-Managed Transaction Demarcation. See also BMTD.

Collaboration Protocol Agreement (CPA) See CPA.

Collaboration Protocol Profile (CPP) See CPP.

Component A component is a grouping of functionality that forms a coherent unit. This unit can be deployed in a component container independently of other components. Applications can then be built by calling on the functionality of multiple, specialist components. J2EE applications are built from various types of component, such as Web Components and EJBs. See also Container.

Connector See JCA (Java Connector Architecture).

Container A container provides services for a component. These services can include lifecycle management, security, connectivity, transactions, and persistence. Each type of J2EE component is deployed into its own type of J2EE container, such as a Web Container or an EJB Container. See also Component.

Container-Managed Persistence (CMP) See CMP.

Container-Managed Relationships (CMR) See CMR.

Container Managed Transaction Demarcation (CMTD) See CMTD.

Cookie A cookie is a short text string sent as part of an HTTP request and response. Because HTTP is a stateless protocol, cookies provide a way of identifying the same client across multiple HTTP requests. Any cookie sent by a server is stored by the client and then submitted whenever another request is made to the same server. Cookies form the basis of most Web-based session management.

CORBA (Common Object Request Broker Architecture) CORBA, from the Object Management Group (OMG), defines a distributed environment consisting of client-server connectivity integrated with a set of distributed services. CORBA clients and servers connect to a local Object Request Broker (ORB) for connectivity and can register and discover each other in the Common Object Services Naming Service (COS Naming). There are many other CORBA services defined, including security, transaction, and persistence.

CPA (Collaboration Protocol Agreement) A CPA is an XML document that defines an agreement between two parties who are using ebXML to conduct e-business. The CPA defines an intersection of the two parties' CPPs, specifying which protocols and mechanisms they will use to exchange information and services. See also CPP.

CPP (Collaboration Protocol Profile) A CPP is an XML document that defines the services and capabilities offered by an organization that provides e-business services using ebXML. See also CPA.

Crimson The Crimson XML parser from Apache is used to provide the XML parsing functionality of the Sun JAXP reference implementation. See also Apache, Xerces.

Custom Tag Library See Tag Library.

Data-Tier See Integration Tier.

Declarative attributes Declarative attributes provide a way for a component to specify requirements to its container by means of attributes defined in the deployment descriptor. These requirements can include when to start and stop transactions and the level of security required by different parts of the component. Delegating control of such functionality to the container and defining them in the deployment descriptor rather than using code means that they are more easily changed when configuring an application.

Deployment Descriptor A deployment descriptor defines metadata for the component or application with which it is associated. J2EE deployment descriptors are XML documents that convey the requirements that a component or application has of its container (such as security requirements). The deployment descriptor can also define the relationships between different classes in the component, naming information, persistence requirements, and so on.

Design pattern See Pattern.

Digital certificate A digital certificate provides a way of signing digital data in such a way that it authoritatively proves the identity of the sender. Certificates are usually issued by trusted third parties called certification authorities.

DNS (Domain Name System) DNS is the mechanism whereby Internet applications can resolve URLs (such as `java.sun.com`) to IP addresses (such as `192.18.97.71`), acting as a basic directory service. It also provides reverse resolution and information on the location of e-mail servers.

Document Object Model (DOM) See DOM.

Document Type Definition (DTD) See DTD.

DOM (Document Object Model) The document object model is an API defined by the W3C for manipulating and traversing an XML document. The API is defined in language-neutral (CORBA) IDL, and Java-based XML parsers provide a Java-language mapping of it. DOM is one of the two main parsing APIs provided by JAXP. See also JAXP, SAX, and W3C.

Domain Layer The term domain layer is sometimes used to denote the group of logical components that provide the data model for an application. These components are manipulated by other components in the business layer to perform business-oriented functionality.

Domain Name System (DNS) See DNS.

DTD (Document Type Definition) The structure of an XML document can be defined using a DTD. The DTD syntax forms part of the XML specification, but is somewhat limited in its descriptive capabilities. For this reason, it is being superseded by XML Schema. See also XML Schema.

EAR (Enterprise Archive) An EAR file contains the components and deployment descriptors that make up an enterprise application. An EAR is the unit of deployment for a J2EE server (that is, how it expects applications to be packaged).

EAI (Enterprise Application Integration) Many existing enterprise applications reside on systems, such as mainframes. Providing connectivity and interoperability with such systems for an n-tier application is commonly termed Enterprise Application Integration. Some n-tier reference models will refer to an integration tier, which is a combination of components that connect to databases and EAI components, sometimes called enterprise information systems. See also EIS.

ebXML (Electronic Business XML) The ebXML initiative has produced a set of e-business standards that range from data transportation through to the choreography of business processes. These standards provide a platform for e-business and a set of value-added services when sending e-business messages. See also CPA, CPP, JAXM.

Electronic Business XML (ebXML) See ebXML.

EIS (Enterprise Information Systems) An enterprise information system is any source of enterprise data, such as a database or mainframe. EIS systems will be accessed through an integration tier. See also EAI.

EJB (Enterprise JavaBean) An EJB is a J2EE business component that lives within a J2EE EJB container. EJBs can be Session beans, Entity beans, or Message-driven beans. An EJB consists of home and remote interface definitions, the bean functionality, and the metadata required for the container to correctly interact with the bean.

EJB Container An EJB container provides services for the EJBs deployed within it. It will control access to the EJB instances and will call the lifecycle methods on each EJB at the appropriate time. The container also provides the persistence and relationship mechanisms used by Entity EJBs.

ejb-jar An `ejb-jar` file contains one or more EJBs together with the deployment descriptor and resources needed by them. The `ejb-jar` is a unit of deployment for EJB business components. See also EJB.

EJB QL (EJB Query Language) EJBs that use CMP must specify the results expected from the various finder methods defined on their home interface. EJB QL provides a container-independent way of doing this by allowing the developer to associate EJB-based queries with the different finder methods.

EJB Query Language See EJB QL.

Enterprise Application An enterprise application consists of one or more J2EE components packaged in an EAR archive. An enterprise application is the end result of a J2EE development.

Enterprise Application Integration (EAI) See EAI.

Enterprise Archive (EAR) See EAR.

Enterprise Information Systems (EIS) See EIS.

Enterprise JavaBean (EJB) See EJB.

Enterprise Resource Planning (ERP) See ERP.

ERP (Enterprise Resource Planning) ERP packages provide pre-packaged, configurable components that provide core enterprise functions, such as personnel management and operations. Such systems are core to enterprise operations, so they must be integrated with other enterprise applications, such as those developed using J2EE. Information from such systems can be retrieved and manipulated through J2EE Connectors or Web Services. See also JCA.

Entity EJB (or Entity Bean) An Entity EJB is a data component used in a J2EE application. Entities frequently map onto domain Entities discovered during analysis and design.

eXtensible Markup Language (XML) See XML.

eXtensible Stylesheet Language (XSL) See XSL.

eXtensible Stylesheet Language Formatting Objects (XSL-FO) See XSL-FO.

eXtensible Stylesheet Language Transformations (XSLT) See XSLT.

Home Interface The home interface of an EJB is a remote factory interface that is registered using JNDI. Clients will discover this interface and use its methods to create, remove, or find one or more EJBs. See also EJB.

HTML (Hypertext Markup Language) HTML is the language used to define Web pages that display in a Web browser, such as Netscape Navigator or Microsoft Internet Explorer. See also HTTP.

HTTP (Hypertext Transfer Protocol) HTTP is the standard transport mechanism used between Web clients and servers. It is used to fetch HTML documents, and also as the underlying transport for Web Services. HTTP servers can be set up on any TCP endpoint, but are usually found on port 80 or common alternatives (8000, 8080, 8888, and so on). See also HTML, HTTPS, and Web Service.

HTTPS (Secure Hypertext Transfer Protocol) HTTPS uses SSL sockets to encrypt HTTP traffic between a client and a server and to authenticate the server to the client (and possibly vice versa). HTTPS uses a different endpoint (443) from standard HTTP. See also HTTP.

HyperText Markup Language (HTML) See HTML.

HyperText Transfer Protocol (HTTP) See HTTP.

IMAP (Internet Message Access Protocol) IMAP is a flexible way of dealing with Internet-based e-mail. Using IMAP, messages stay in a user's mailbox on the server while the client retrieves metadata about them (such as the size, sender, and so on). The client can then selectively download messages, rather than having to download all e-mails in one go. See also POP.

Initial Context A JNDI initial context is the starting point for JNDI interaction. J2EE components will obtain an initial context to discover resources, properties, and other J2EE components.

Integration Tier The set of machines on which the data access components execute in an n-tier or 3-tier application. See also n-tier, 3-tier.

Internet Message Access Protocol (IMAP) See IMAP.

J2EE Java 2 Enterprise Edition.

J2EE application See Enterprise Application.

J2EE Component A J2EE component is the basic unit of a J2EE application. Different components will serve different purposes, such as presentation of data, provision of business logic, or access to underlying data. See also Web Component.

J2EE Container A J2EE container provides the services and environment required by a particular type of J2EE component. See also Web Container.

J2EE Pattern A J2EE Pattern is a pattern that is implemented using J2EE technologies. J2EE Patterns can help to improve the quality of the J2EE applications within which they are applied. See also Pattern.

J2EE Reference Implementation (RI) See J2EE RI.

J2EE RI (J2EE Reference Implementation) The J2EE RI (or the Java 2 SDK Enterprise Edition—J2SDKEE) serves as a proof-of-concept for the technologies defined in the J2EE specification. The team that produces the JSR for each version of J2EE is responsible for delivering a reference implementation for it. The J2EE RI is freely downloadable and provides a useful test environment for J2EE developers.

J2EE Server A J2EE server is the underlying J2EE platform that provides the services required by J2EE containers. J2EE servers are typically delivered in the form of application servers. See also J2EE Container.

Jakarta Project The Jakarta Project is the overarching project for Java-oriented development at the Apache Foundation. This includes the Tomcat servlet and JSP engine. See also Apache.

JAF (JavaBeans Activation Framework) JAF provides a way of associating a particular MIME type with an application or component that knows how to process data of that MIME type. JAF is used in various parts of J2EE including JavaMail and Web Services.

JAR (Java Archive) JAR files are a compressed archive format in which Java classes and associated resources are typically stored. All of the various archives used by J2EE are delivered in JAR files. See also EAR , ejb-jar, WAR.

Java API for XML Messaging (JAXM) See JAXM.

Java API for XML Parsing (JAXP) See JAXP.

Java API for XML Registries (JAXR) See JAXR.

Java API for XML-based RPC (JAX-RPC) See JAX-RPC.

Java Architecture for XML Binding (JAXB) See JAXB.

Java Archive (JAR) See JAR.

JavaBean A JavaBean is a Java class that conforms to certain rules on method naming, construction, and serialization. JavaBeans are used within J2EE to contain Java functionality and data in Web components or as data carriers between layers. Note that a JavaBean is not an EJB. See also EJB.

JavaBeans Activation Framework (JAF) See JAF.

Java Connector Architecture (JCA, Connectors) See JCA.

Java Community Process (JCP) See JCP.

Java Database Connectivity (JDBC) See JDBC.

Java Data Objects (JDO) See JDO.

Java IDL Java IDL is the delivery mechanism for CORBA IDL support under Java. The Java IDL compiler allows you to compile CORBA IDL into Java stubs and skeletons that can be used to communicate with remote CORBA objects. See also CORBA.

JavaMail JavaMail is part of the J2EE platform that allows you to send and receive e-mail messages.

Java Message Service (JMS) See JMS.

Java Naming and Directory Interface (JNDI) See JNDI.

JavaServer Pages (JSP) See JSP.

JavaServer Pages Standard Tag Library (JSPTL) See JSPTL.

Java Transaction API (JTA) See JTA.

Java Transaction Service (JTS) See JTS.

Java Web Service (JWS) file See JWS.

JAXB (Java Architecture for XML Binding) JAXB defines an architecture for marshalling data between Java and XML formats. It provides tools to convert XML DTDs and Schemas into Java classes.

JAXM (Java API for XML Messaging) JAXM defines how J2EE components send and receive XML-based messages over SOAP. JAXM will form part of J2EE 1.4. See also SOAP.

JAXP (Java API for XML Processing) JAXP defines the interfaces and programming model for the parsing, manipulation, and transformation of XML in Java.

JAX Pack (or Java XML Pack) The JAX Pack provides an interim delivery mechanism for the various XML-related APIs currently under development. After these APIs have been incorporated into J2EE 1.4, the JAX Pack may disappear. See also JAXM, JAXP, JAXR, JAX-RPC.

JAXR (Java API for XML Registries) JAXR defines how a J2EE component will access and manipulate XML data held in XML-oriented registries, such as UDDI and the ebXML registry and repository.

JAX-RPC (Java API for XML-based RPC) JAX-RPC defines how J2EE components make and receive XML-based RPC calls over SOAP. It specifies the relationship between the definition of an interface in WSDL and the Java binding for that interface. See also SOAP, UDDI.

JCA (Java Connector Architecture) The JCA defines how external data sources, such as ERP systems, should be made available to J2EE components. The JCA mechanism is very similar to the JDBC standard extension that allows components to discover and use data sources defined by the container.

JCP (Java Community Process) The JCP is the process under which vendors and individuals in the Java community work together to create and formalize the APIs and technologies used in the Java platform. Standardization efforts under the JCP are referred to as Java Specification Requests (JSRs). See also JSR.

JDBC (Java Database Connectivity) JDBC provides data access for Java applications and components.

JDO (Java Data Objects) JDO specifies a lightweight persistence mechanism for Java objects. Its functionality lies somewhere between Java Serialization and entity EJBs.

JMS (Java Message Service) JMS specifies how a J2EE component can produce or consume messages by sending them to, or retrieving them from, a queue or a topic. JMS supports both the point-to-point model and the publish/subscribe model of message passing.

JNDI (Java Naming and Directory Interface) JNDI provides a generic API for access to information contained in underlying naming services, such as the CORBA CoS Naming service. JNDI is used by J2EE components to discover resources, configuration information, and other J2EE components.

JSP (JavaServer Pages) JSPs provide a model for mixing static tagged content, such as HTML or XML, with content dynamically generated using Java code. Such code can be embedded in the page itself or, more usually, encapsulated in a JavaBean or tag library. See also JavaBean, Tag Library.

JSP Directive Directives are messages to the JSP container that are embedded in a JSP page. They appear as tags delimited by `<%@ %>`. Directives include `page`, `taglib`, and `include`. A common example would be a page directive that defines the error page for this JSP. See also JSP, JSP Error Page.

JSP Error Page An error page can be defined for each JSP page to handle any errors occurring on that page. Any unhandled exceptions will cause the error page to be displayed. The same error page can be shared between multiple JSP pages. An error page is basically a JSP page that has access to error-specific information, such as the exception that caused the error. See also JSP.

JSP Expression A JSP expression is a Java statement that is executed, and the result of this statement is coerced into a string and output at the current location in the page. Expressions are delimited by `<%= %>`. A typical example would use a Java statement to evaluate the current date or time to be inserted at the current location. See also JSP.

JSP Scriptlet A JSP scriptlet is a section of Java code embedded in a JSP page and delimited by `<% %>`.

JSPTL (JavaServer Pages Standard Tag Library) The JSPTL is an initiative under the JCP to create a standard tag library for common functionality, such as conditional evaluation and looping. See also Tag Library.

JSR (Java Specification Request) A JSR is a project under the auspices of the JCP that defines a new standard for a Java API or technology. JSRs are run by groups of experts drawn from vendors and the broader Java community. See also JCP.

JTA (Java Transaction API) The JTA defines an API that allows J2EE components to interact with transactions. This includes starting transactions and completing or aborting them. The JTA uses the underlying services of the JTS. See also JTS.

JTS (Java Transaction Service) The JTS is a low-level, Java-based mapping of the CORBA Object Transaction Service. Transaction managers that conform to JTS can be used as part of a J2EE environment to control and propagate transactions on behalf of J2EE components. J2EE components will not use JTS directly, it is used on their behalf by their container or through the JTA. See also JTA.

JWS (Java Web Service file) A JWS file is a Java class file with a `.jws` extension. When placed under the appropriate part of the Apache Axis hierarchy, this Java class will automatically be exported as a Web Service. See also Axis.

Kerberos Kerberos is a strong, distributed security mechanism that uses encryption and signed "tickets" to allow clients and servers to interoperate in a secure manner.

LDAP (Lightweight Directory Access Protocol) LDAP is a standard protocol for accessing data in a directory service. Common directory services such as Microsoft's Active Directory and Novell's NDS support LDAP. JNDI can be used to deliver LDAP requests using the LDAP service provider. See also Active Directory, JNDI, NDS.

Lightweight Directory Access Protocol (LDAP) See LDAP.

Local Home Interface A local home interface is an EJB factory interface that returns EJB local interfaces. The local home interface is for use by clients that run in the same server and reduces the overhead associated with remote RMI calls. See also Home Interface, Local Interface.

Local Interface A local interface is a business- or data-access interface defined by an EJB that is intended to be used by clients running in the same server. Using a local interface reduces the overhead associated with remote RMI methods. Local interfaces form the foundation for container-managed relationships used by Entity EJBs. See also CMR, Local Home Interface, Remote Interface.

MDB (Message-Driven EJB) A Message-driven bean is an EJB that processes JMS messages. The bean implements an `onMessage` method, just like a JMS message consumer. Messages delivered to the associated queue will be passed to the MDB's `onMessage` method, so the MDB will be invoked asynchronously (the client could be long gone).

Message-Driven EJB (or Message-Driven Bean, MDB) See MDB.

Microsoft SQL Server Microsoft SQL Server is Microsoft's flagship enterprise database. SQL Server is now part of Microsoft's .NET server range. J2EE components can access data in a SQL Server database through standard data access mechanisms, such as JDBC. See also JDBC.

MIME (Multipurpose Internet Mail Extensions) MIME provides a way of defining a multi-part message in which each part contains a different type of data. Within the message, each part has its own MIME header defining the content type of that part and delimiting that part from the other parts. Common uses of MIME include Internet e-mail and SOAP.

Multipurpose Internet Mail Extensions (MIME) See MIME.

N-Tier Modern distributed applications are defined in terms of multiple tiers. A 3-tier application has three physical tiers containing presentation, business, and data access components. In reality, applications can have many more physical tiers, each of which can be some specialization of the three tiers listed, or as a representation of ultimate clients and data sources. As such, these applications are referred to as n-tier to indicate that there are a variable number of tiers (3 or more). See also 3-tier.

NDS (Novell Directory Services) NDS is Novell's popular directory service, originating from its NetWare family of products.

Novell Directory Services (NDS) See NDS.

OASIS (Organization for the Advancement of Structured Information Standards) OASIS is a non-profit, international consortium that creates interoperable industry specifications based on public standards, such as XML and SGML. OASIS is one of the sponsors of ebXML. See also ebXML.

Object-Oriented Database Management System (OODBMS) See OODBMS.

Object Relational Database Management System (ORDBMS) See ORDBMS.

OODBMS (Object-Oriented Database Management System) An OODBMS provides persistent storage that supports the OO paradigm so that data definition can be done in terms of classes, inheritance, and methods. Data retrieval can be performed in terms of object instances in contrast to record sets. See also ORDBMS.

Oracle Oracle produce several J2EE-related products. The Oracle database can be used as an enterprise-class data store as part of a J2EE application. The Oracle application server is itself a J2EE application server that can host a J2EE-compliant application. As you would expect, this gives performance and functionality benefits when combined with Oracle's database.

ORDBMS (Object Relational Database Management System) An ORDBMS provides an OO mapping on top of a traditional relational database. This means that the developer can work in terms of objects and classes, and the ORDBMS takes the responsibility for mapping these classes and objects to the underlying database tables. See also OODBMS.

Organization for the Advancement of Structured Information Standards (OASIS) See OASIS.

Pattern A pattern is a solution to a problem in a given context. Patterns commonly occur in software design and architecture. By using patterns, designers and architects can improve the quality of the software and systems they produce.

Post Office Protocol (POP/POP3) See POP.

POP/POP3 (Post Office Protocol) POP defines a way for an e-mail client, such as Eudora, to retrieve messages from a mailbox maintained by an e-mail server. All messages must be downloaded before they can be examined or read. See also IMAP, SMTP.

Presentation Layer The presentation layer is a term used to refer to the logical layer containing the interaction with the user of the application. For a Web-based application, this typically means the generation of HTML or XML by servlets and/or JSPs. For an application client, the presentation is typically done through a Swing GUI.

Presentation Tier The set of machines on which the presentation components execute in an n-tier or 3-tier application. See also 3-tier, N-tier.

Reference Implementation See J2EE RI.

Remote Interface The business or data access methods exposed by an EJB are referred to as its remote interface. The interface extends the RMI `Remote` interface, indicating that it is to be used outside of the current virtual machine. Each EJB has one remote interface, and this is the type returned by finder and creator methods on the EJB's home interface.

Remote Method Invocation (RMI) See RMI.

Remote Procedure Call (RPC) See RPC.

Remote Reference A remote reference is an object reference that refers to a remote object. The method calls made through the remote reference will be propagated to the remote object using RMI. In J2EE terms, a remote reference will usually refer to an EJB or its home interface and will be retrieved from a finder/creation method or through JNDI, respectively. See also RMI.

RI See J2EE RI.

RMI (Remote Method Invocation) RMI is a Java-based, object-oriented RPC mechanism. All communication with EJBs in a J2EE application is done via RMI (except for Message-driven beans). RMI defines a syntax and mechanism for accessing remote Java objects and also for passing serialized Java objects between client and server. See also RMI-IIOP.

RMI-IIOP RMI-IIOP defines a way that RMI RPC calls can be carried over the CORBA IIOP transport. This allows for interoperability between different J2EE application servers as well as between RMI clients and servers and CORBA clients and servers. See also CORBA, RMI.

RPC (Remote Procedure Call) An RPC is a method call that spans processes, frequently across a network. A client-side stub (or proxy) and a server-side skeleton (or stub) will make the issuing of RPCs look similar to a local method call. See also CORBA, RMI.

SAX (Simple API for XML) SAX was defined by members of the XML-DEV e-mail list (and formalized by David Megginson) as a way of processing XML in Java. The SAX API is event driven, notifying the Java program as XML elements and content are encountered. SAX is generally regarded as lighter-weight than the DOM API and is delivered as part of JAXP. See also DOM, JAXP.

Scriptlet See JSP Scriptlet.

Secure HyperText Transfer Protocol (HTTPS) See HTTPS.

Secure Sockets Library (SSL) See SSL.

Servlet A servlet is a Web component that is written entirely in Java. Servlets have a defined lifecycle and allow Web developers to consume (most commonly) HTTP requests and generate responses. Responses can be in the form of HTML, XML, or any text or binary format. See also JSP, Web Component.

Session EJB (or Session Bean) Session EJBs are intended to house business logic and processing in a typical J2EE application. Session EJBs will provide business services to the presentation tier and will use Entity EJBs, connectors, or direct database access to retrieve business data.

Session EJBs can be either stateful (they retain state between invocations) or stateless (state is not retained between invocations).

SGML (Standard Generalized Markup Language) SGML is a forerunner of both HTML and XML. It is a very flexible general purpose markup language that, like XML, can be used to mark up any form of data. However, its flexibility leads to it being somewhat unwieldy. The originators of XML intended to keep much of the flexibility of SGML while deriving a simpler syntax. See also XML.

Simple API for XML (SAX) See SAX.

Simple Mail Transfer Protocol (SMTP) See SMTP.

Simple Object Access Protocol (SOAP) See SOAP.

SMTP (Simple Mail Transfer Protocol) The SMTP standard defines how e-mail servers send messages to each other. SMTP forms the backbone of the Internet e-mail delivery system. See also IMAP, POP.

SOAP (Simple Object Access Protocol) SOAP is an XML-based, de-facto standard for the encoding of XML-based messages. The messages can be intended as method names and parameters for a remote procedure call, or as an XML-encoded message to be processed and potentially passed on. SOAP is used as the underlying transport for all Web Services. SOAP is being formalized under the auspices of the W3C. See also JAX-RPC, JAXM, SOAP-RP, Web Service.

SOAP Routing Protocol See SOAP-RP.

SOAP-RP (SOAP Routing Protocol) SOAP-RP is an evolving standard that adds the ability to route SOAP messages. The original SOAP specification only dealt with SOAP messages sent between two parties. A fully-functional messaging system should be able to support multi-hop messages. This is the intention of SOAP-RP. See also SOAP.

SQL (Structured Query Language) SQL is a language used to create, update, retrieve, delete, and manage data in a relational database. SQL statements are defined from simple selection of data through to the invocation of parameterised stored procedures. Although the core SQL statements are standardized, some vendors provide their own extensions. See also ANSI SQL, ANSI SQL 92.

SQL 92 See ANSI SQL 92.

SQLJ SQLJ defines a way of embedding SQL statements in Java code and, as such, is an alternative to the use of JDBC. SQLJ also defines how Java code can be used to create stored procedures. SQLJ is a vendor-independent initiative. More information can be found online at `http://www.sqlj.org`.

SSL (Secure Sockets Library) SSL defines a standard way of using encryption across a sockets connection. This includes authentication of the server (and optionally the client) using digital certificates, and the exchange of encryption keys. SSL is commonly used as the basis for transporting secure versions of higher-level protocols, such as secure HTTP (HTTPS). See also HTTPS.

Standard Generalized Markup Language (SGML) See SGML.

Stateful Session EJB (or Stateful Session Bean) See Session EJB.

Stateless Session EJB (or Stateless Session Bean) See Session EJB.

Structured Query Language (SQL) See SQL.

Sybase Sybase is one of the major server vendors. Their products include the Adaptive Server database and the EAServer application server.

Tag Library A tag library defines a set of XML-compliant tags that can be used as part of a JSP. Each tag is associated with a particular piece of Java code. When the JSP processor encounters one of these tags, it will invoke the associated Java code. The Java code may generate new content, or it may perform other tasks such as looping or access to J2EE resources. See also JSP.

Taglib See Tag Library.

Tomcat The Tomcat servlet engine is an open-source Java implementation delivered by the Apache Foundation. Tomcat (and the associated Jasper JSP engine that is delivered with it) have formed the reference implementation for servlets and JSPs. Tomcat can run as a standalone server, or it can be plugged into most Web servers, including the Apache Web Server. See also Jakarta Project.

UDDI (Universal Description, Discovery and Integration) UDDI is one of the main technologies used for registration and discovery of Web Services. UDDI defines a set of SOAP messages that can be used to access XML-based data in a registry. It also defines a registry information model to structure the data stored and make it easier to search and navigate.

UDDI4J UDDI4J is a Java-based API from IBM that provides Java wrappers for the UDDI SOAP messages. This API fills the same role as JAXR. See also JAXR, UDDI.

UML (Unified Modelling Language) UML defines a largely diagrammatic language for capturing system requirements and expressing system design in object-oriented terms. UML diagrams are commonly used to illustrate class relationships and object interactions. UML was created from a merger of previous OO methodologies, including Booch, Jacobsen, and OMT.

UML Class Diagram A UML class diagram represents a domain entity of some form that has usually been discovered by analysis. This entity can represent information, functionality, or both. An example would be a `Customer` class that had attributes and functionality associated with the role of a customer. Class diagrams can also be used to represent the relationships between different classes in a system. See also UML.

UML Collaboration Diagram A UML collaboration diagram is a way of showing interactions between objects at the same time as showing the relationships between the objects. It combines some of the features of a class diagram with some of the features of a sequence diagram. See also UML.

UML Component Diagram A UML component diagram shows the components in a system and their dependencies. See also UML.

UML Interaction Diagram The term "UML interaction diagram" refers to any one of several forms of UML object-based diagrams that show the interactions between objects, such as sequence diagrams. See also UML Sequence Diagram.

UML Sequence Diagram A UML sequence diagram shows the interactions between two or more objects over time. Each object is represented by a "swim lane" down the page, and messages are shown passing to and fro. See also UML.

Unified Modelling Language (UML) See UML.

Uniform Resource Identifier (URI) See URI.

Uniform Resource Locator (URL) See URL.

Universal Description, Discovery and Integration (UDDI) See UDDI.

URI (Uniform Resource Identifier) A URI is a string-based name for an abstract or physical resource. The URI specification defines how specific resource identifiers, such as URLs, should be formatted. If you like, a URI is the "abstract base class" of other resource identifiers, such as URLs. See also URL.

URL (Uniform Resource Locator) A URL is a string-based name for identifying a resource available across the Internet. An absolute URL begins with the protocol (`http`), then a colon (`:`), and then the specific address using that protocol. This usually contains a hostname, a relative path, and possibly other components. An example of a URL is `http://java.sun.com`. See also URI.

Validation See XML validation.

W3C (World Wide Web Consortium) The W3C is a vendor-neutral body created in 1994 by Tim Berners-Lee to lead the development of common Web protocols. The W3C has more than 500 Member organizations from around the world.

WAP (Wireless Access Protocol) WAP is a standard protocol for transporting data between a mobile device and a server. Most WAP servers take the form of gateways that provide onward access to Internet resources. The WAP protocol layers take the place of TCP/IP, and are designed to allow for the unpredictability of mobile connectivity. See also WML.

WAR (Web Archive) A WAR file is the unit of deployment for one or more Web components. A WAR file is a JAR-format file that contains servlets and/or JSPs together with deployment information and additional resources required by the component. See also EAR, JAR.

Web Application A Web application provides functionality accessible over the Web, usually from a Web browser. A Web application can be delivered in a WAR file and deployed under a compliant Web container. See also Enterprise Application, WAR.

Web Archive (WAR) See WAR.

Web Component A Web component is a unit of functionality that forms part of a Web application or Enterprise application. A Web component consists of one or more JSPs and/or servlets together with deployment information and additional resources required by the component. A Web component is deployed into a Web container that controls its access to resources and its lifecycle. See also Enterprise Application, WAR, Web Application, Web Container.

Web Container A Web container provides services, such as access control, resource access, and lifecycle management for one or more Web components. See also Web Component.

Web Service A Web Service is a programmatic interface for functionality accessible using Web protocols. Web Services are accessed over SOAP and may be registered in Web Service registries, such as UDDI. The functionality of a Web Service is usually defined in terms of WSDL. See also ebXML, SOAP, UDDI, WSDL.

Web Service Registry A Web Service registry is an XML-based repository for information about Web Services. Service providers will register their services in such a repository and clients will search for required services there. Examples of Web Service Registry standards include UDDI and the ebXML Registry and Repository standard. See also ebXML, UDDI.

Web Services Description Language (WSDL) See WSDL.

Web Tier See Presentation-Tier.

Well-formed (of an XML document) An XML document is said to be well formed if it obeys certain rules regarding structure laid down in the XML standard. XML documents that are not well formed cannot be manipulated by XML tools, such as parsers, and will generate errors when processed. See also XML, XML Validation.

Wireless Access Protocol (WAP) See WAP.

Wireless Markup Language (WML) See WML.

WML (Wireless Markup Language) WML is a markup language, similar to HTML, that is targeted at mobile devices. Due to the limited nature of most mobile displays, WML is far less feature-rich than HTML. WML documents are delivered to mobile devices over WAP. See also WAP.

World Wide Web Consortium (W3C) See W3C.

WSDL (Web Services Description Language) WSDL is an XML syntax for describing a Web Service interface, the protocols through which that interface can be reached, and the location of one or more servers that implement the interface. See also Web Service.

WSDL4J WSDL4J is a Java-based API for WSDL manipulation defined by IBM. This API occupies the same role as the forthcoming API for WSDL being defined by JSR 110. See also WSDL.

Xalan Xalan is an open-source XSLT processor from the Apache Foundation that is written in Java and is accessible from Java. Xalan supports the TrAX API for Java-based XML transformations. Xalan is used by the JAXP reference implementation as the basis for its XSLT transformation support. See also Apache.

Xerces Xerces is an open-source XML parser from the Apache Foundation that is written in Java and accessible from Java. Xerces supports the SAX and DOM APIs and is very popular in the Java community. It is likely that Xerces will soon be merged with Crimson to create a single Apache XML processor. See also Apache, Crimson.

XML (eXtensible Markup Language) XML is a tag-based syntax for adding information into text documents. XML does not define any specific tags, rather it defines the structure and conventions to be used by custom tags created for a variety of purposes. XML documents consist of a set of elements (delimited by opening and closing tags) and optionally attributes on those elements. The XML specification is defined and maintained by the W3C. See also W3C.

XML Schema The XML Schema standard defines an XML grammar that can be used to define the contents of an XML document. An XML schema can define the data types expected, sequencing of XML elements, the presence and values of attributes, and so on. XML schemas are associated with XML documents using namespaces. See also XML.

XML validation A valid XML document is a well-formed XML document that conforms to a particular DTD or XML Schema. A parser can be asked to validate an XML document against a DTD or schema and will generate errors if the structure checking fails. See also DTD, Well-formed, XML Schema.

XPath The W3C XPath standard describes a syntax for selecting parts of an XML document. XPath is used by XSLT to identify which parts of the source document are to be transformed by a particular rule. See also XSLT.

XPointer The W3C XPointer standard describes how a fragment of an XML document can be identified using a URI combined with XPath syntax. This is similar in concept to an HTML-based URL that includes an anchor (for example `http://www.tempuri.org/usefulfacts.html#WEBSERVICES`) to locate a specific part of the document. See also XPath.

XSL (eXtensible Stylesheet Language) The W3C XSL standard covers both XSLT and XSL-FO. Originally, there was intended to be a single XML-oriented style sheet language (such as is the case with the Cascading Style Sheet Language for HTML), but two distinct elements of functionality were identified (transformation and rendering), so two separate standards were spawned. See also XSL-FO, XSLT.

XSL-FO (eXtensible Stylesheet Language Formatting Objects) XSL-FO defines a set of XML-based formatting objects that can be used to render a document. XSL-FO is a more generic rendering format than, for example, HTML, and can be used on a wider variety of devices and applications. See also XSL.

XSLT (eXtensible Stylesheet Language Transformations) XSLT is a declarative language and processing model used to convert one dialect of XML into another dialect of XML (or some other text-based format). XSLT is frequently used when importing or exporting business documents in XML format. See also XSL.

INDEX

Symbols

& (ampersand), 990
<> (angle brackets), 509, 558, 990
*** (asterisk), 993**
, (comma), 993
— (double hyphens), 710
“ (double quotes), 101, 990
/ (forward slash), 100-101
() (parenthesis), 993
% (percent sign), 565
| (pipe character), 993
+ (plus sign), 993
? (question mark), 993
‘ (single quote), 101, 990
<!— —> tag
  HTML (Hypertext Markup Language), 511
  XML (Extensible Markup Language), 709
100-199 status codes (HTTP), 508
2PC (two phase commit) protocol, 354-356, 359-360
2-tier design
  disadvantages, 12-13
  layers, 11-12
3-tier development. *See* *n*-tier development
200-299 status codes (HTTP), 508
300-399 status codes (HTTP), 508
400-499 status codes (HTTP), 508
500-599 status codes (HTTP), 508
A
<A> tag (HTML), 511
absolute URLs (Uniform Resource Locators), 475
abstract accessor methods, 274-275
abstract classes, 273
abstract schema names, 294
accessing data, 22
  data access logic, 12
  Data Access Object pattern, 804-806
    createJob() method, 810-811
    DirectJobDAOImpl implementation, 807-809
    JobDAO interface, 806
    JobValueObject object, 806-807
  EJB services, 131
  servlet variables, 575
  UDDI (Universal Description, Discovery, and Interaction)
    JAXR (Java API for XML Registries), 934-937
    locally hosted registries, 929
    public production registries, 929
    public test registries, 929
    UDDI4J, 929-932
    WSKT Client API, 932-934
ACID test, 336
acknowledgements
  AUTO_ACKNOWLEDGE, 437
  DUPS_OK_ACKNOWLEDGE, 438
actions, 558
activating Entity EJBs (Enterprise JavaBeans), 235

activations (Sequence diagrams), 973
Active Directory, 19
actors, 967
adapters
Adapter classes, 204-205
resource adapters, 835-836
add() method, 287
addAll() method, 287
addAttachmentPart() method, 951
addBodyElement() method, 948
addBodyPart() method, 477-478, 482
addChildElement() method, 948
addCookie() method, 533
addElement() method, 736
addNamingListener() method, 120
addRecipient() method, 469
addresses (e-mail)
HTML e-mail, 473
recipient addresses, 469
sender addresses, 469
admin.jsp file, 815
administered objects, 399
administration tool, 65
Advertise interface, 801-802
advertise.jsp page, 592-594, 632-634, 693
AdvertiseValueObject object, 803
afterBegin() method, 352
Agency case study, 72-76, 585, 734
Agency directory, 76
AgencyTable servlet, 546-552
CaseStudy directory, 76
CMP support, 276-277
customer list, 644
database installation, 77-78
declarative authorization
network security requirements, 689-690
roles, 685
security constraints, 686-691
deploying, 156-157
ERD diagram, 72-73
examining, 154-155
Examples directory, 76
Exercise directory, 76-77
jobSummary document
attributes, 708-709
code listing, 708
DTD (document type declaration), 713
namespace, 714-715
XML Schema, 716-717
Message-driven beans, 447
AgencyBean, 449-450, 735-736
ApplicantMatch bean, 737-739
ApplicationMatch bean, 451-456
deploying, 456-457
MessageSender class, 736-737
queues, 456
RegisterBean, 449-450
sender helper class, 447-449
testing, 457
one-to-many relationships, 284
patterns, 797-798, 822
client-side proxies and delegates, 820-821
data access without EJBs (Enterprise JavaBeans), 804-811
data exchange and Value Objects, 800-804
entity creation, 812-813
JSP (JavaServer Page) creation, 813-817
JSPs (JavaServer Pages) and separation of concerns, 817-820
messages and asynchronous activation, 811
service location, 821-822
Session Facades, 798-800
programmatic authorization
advertise.jsp customer name selection, 693
agency.jsp customer options, 692
role references, 693-694
programmatic security, 682
AgencyBean.java, 681
ejbCreate() method, 679
role references, 680
roles, 675, 686
Solution directory, 76
testing, 158-160
troubleshooting, 160-161
Web interface
advertise.jsp page, 592-594, 632-634, 693
agency.css style sheet, 589
agency.jsp page, 589-590, 692, 814
agency.ldif configuration file, 105-106
AgencyBean.java, 582-584
agencyName.jsp page, 581-582
dateBanner.jsp page, 570
deploying, 597-600
EJB references, 598
errorPage.jsp, 595-597
look and feel, 588-592
name.jsp page, 572-573

portal page, 587
skills.jsp, 627-628
structure and navigation, 585-587
table.jsp page, 576-577
tableForm.jsp page, 576
updateCustomer.jsp, 594-595
XML Schema, 734
Agency directory, 76
Agency EJG, 133
Agency Session bean, 172
agency.css style sheet, 589
agency.jsp page, 589-590, 692, 814
agency.ldif configuration file, 105-106
AgencyBean bean, 449-450, 582-584, 735-736, 817-818
agencyName.jsp page, 581-582
AgencyTable servlet, 546-552
Alexander, Christopher, 788-790
algorithms
checksums, 660
message digests, 660
symmetric encryption, 657-658
Allaire ColdFusion, 24
ALTER TABLE statement (SQL), 978-979
ampersand (&), 990
analysis patterns, 791
ancestor axis (XPath), 763
angle brackets (<>), 509, 558, 990
anonymous security, 121
ANSWERED flag (e-mail), 489
ANY keyword, 712
Apache
Axis toolkit, 877, 881-883
Jakarta Project, 604, 646
appendChild() method, 731
applets
applet clients, 51
JNDI (Java Naming and Directory Interface) parameters, 90
ApplicantMatch bean, 737-739
applicantXML() method, 736
Application Assemblers, 63
Application Component Providers, 63
Application Deployers, 64
application development, 9-10
Enterprise Computing Model, 17-18
lifecycle, 18
naming, 18-19
persistence, 18
security, 19-20
transactions, 19
monolithic development
disadvantages, 10-11
structure, 10
transitioning to *n*-tier, 26
n-tier design, 13-14, 28, 38
advantages, 16-17
business tier, 39-44
client tier, 49-54
component frameworks, 15-16
modularity, 14-16
presentation tier, 44-49
transitioning to, 26
two-tier design
disadvantages, 12-13
layers, 11-12
Application object, 575
application scope, 618
Application Server Enterprise Edition (iPlanet), 24
application-level exceptions, 351
application.xml file, 146
ApplicationMatch bean, 451
code listing, 453-456
deleteByApplicant() method, 452
deploying, 456-457
ejbCreate() method, 451
ejbRemove() method, 451
findByLocation() method, 452
getLocation() method, 452
getSkills() method, 453
InitialContext interface, 451
onMessage() method, 452
skillMatch counter, 453
testing, 457
applications. *See also specific application names*
B2B (business to business), 129
deploying, 66-67
deployment descriptors, 67-68, 526-527
EARs (Enterprise Application Archives), 67
EJB-JAR files, 69
filters, 538-541
listeners, 543-545
Tag Libraries, 612-614
WAR (Web Archive) files, 70
developing. *See* application development
EAI (Enterprise Application Integration), 129
EJB-based, 126
enterprise applications
component relationship descriptions, 146-147
deploying, 193, 322-323
exceptions, 179
J2EE Blueprints, 23-24

JNDI (Java Naming and Directory Interface) properties, 89
identity, 386
modeling, 127
packaging, 66-67
deployment descriptors, 67-68
EARs (Enterprise Application Archives), 67
EJB-JAR files, 69
WAR (Web Archive) files, 70
servlets
accessing, 518
code listing, 514-515
deploying, 515-518
HttpServletRequest interface, 515
HttpServletResponse interface, 515
thick clients, 129
Web applications
deployment descriptors, 526-527
directory structure, 525-526
EJBs (Enterprise JavaBeans), 128
applying patterns, 793-794
case study analysis, 797-798, 822
client-side proxies and delegates, 820-821
data access without EJBs (Enterprise JavaBeans), 804-811
data exchange and Value Objects, 800-804
entity creation, 812-813
JSP (Java ServerPage) creation, 813-817
JSPs (Java ServerPages) and separation of concerns, 817-820
messages and asynchronous activation, 811
service location, 821-822
Session Facades, 798-800
refactoring, 794, 822-823
architectural patterns, 790
archives
EARs (Enterprise Application Archives), 67
WAR (Web Archive) files, 70, 909
ARRAY data type, 367
<assembly-descriptor> tag, 340, 668
associations, 969-970
asterisk (*), 993
asymmetric encryption, 658-659
asynchronous message-based Web Services, 939
asynchronous messaging
exception handling, 415
main() method, 415
MessageListener interface, 414
onMessage() method, 414-415
setMessageListener() method, 414
atomic units, 336
attachments (e-mail)
creating, 482-483
retrieving, 490
processPart() method, 491
RetrieveAttachment application, 492-494
writeFile() method, 491
writeTo() method, 492
sending, 483-485
SOAP messages, 951-952
<attribute> tag, 616
attributes
Class diagrams, 970-971
DTDs (document type declarations)
attribute types, 712-713
default values, 713
defining, 712
LDAP (Lightweight Directory Access Protocol)
defined, 102
modifying, 112-114
reading, 108-109
XML (Extensible Markup Language), 708-709
declaring, 993-994
lookup tag example, 615-617
processing, 765-766
transforming, 770-771, 774-776
validating, 635-637
XPath, 764
AuditFilter servlet
code listing, 537-538
doFilter() method, 544-545
audits
auditing filter
code listing, 537-538
doFilter() method, 544-545
defined, 656
AuthenticateRetrieveMail application, 495-497
authentication
Basic authentication, 683-685
client authentication, 655
defined, 19, 654
Digest authentication, 683
Digest MD5, 696

external, 122, 696
forms-based authentication, 683
GSSAPI, 696
HTTPS client authentication, 684
initial identification, 654
JAAS (Java Authentication and Authorization Service), 58
JavaMail
AuthenticateRetrieveMail application, 495-497
Authenticator class, 494
MyAuthenticator class, 494-495
PasswordAuthentication object, 495
LDAP (Lightweight Directory Access Protocol), 696
SASL (Simple Authentication and Security Layer)
jndi.properties file, 697-698
ListSASL.java example, 696-697
secure authentication schemes, 694
user credentials, 655
authentication property (JNDI), 121, 695
Authenticator class, 494, 499
authorization
declarative authorization
network security requirements, 689-690
roles, 685
security constraints, 686-691
defined, 19, 655
JAAS (Java Authentication and Authorization Service), 58
programmatic authorization
Agency case study, 692-694
getRemoteUser() method, 692
getUserPrincipal() method, 691
isUserInRole() method, 691
AUTO_ACKNOWLEDGE message, 437
Axis toolkit, 877, 881-883

B

B2B (business to business) applications, 129
Bacchus Normal Form (BNF), 293
backslash (\), 101
Basic HTTP (Hypertext Transfer Protocol) authentication, 683-685
BEA Weblogic Server, 24
bean class, 431
bean-managed persistence (BMP), 217
bean-managed transactions, 436-437
deployment, 349
restrictions, 345
Session EJBs (Enterprise JavaBeans), 345-349
BeanOrderServer.java file, 915
beans
creating, 579
declarative security
method permissions, 670-674
role mappings, 674-676
roles, 666-668
security identity, 668-670, 676-678
defined, 578
EJBs (Enterprise JavaBeans). *See* EJBs
parts of, 134
initializing, 581
JAF (JavaBeans Activation Framework), 59
programmatic security, 678
Agency case study, 679-682
getCallerPrincipal() method, 678
isCallerInRole() method, 678
properties
retrieving, 579-580
setting, 580-581
security, 666
BeanSerializers, 912-919
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
beforeCompletion() method, 352
begin() method, 346, 845
beginTransactionIfRequired() method, 349
bidirectional assocation, 970
<BIG> tag (HTML), 511
bin directory, adding to search path, 30
bind() method, 91, 853
binding objects, 90-91
bind() method, 91
example, 91
name persistence, 92
potential problems, 91-92

rebinding, 92
unbinding, 92-93
bindingTemplate structure, 928
BLOB data type, 366, 375
Blueprints, 23-24
BMP (bean-managed persistence), 217
configuring, 248-252
defining interfaces, 225-230
implementing, 231-248
lifecycle, 220-224
obtaining references, 252
BNF (Bacchus Normal Form), 293
<BODY> tag (HTML), 511
body types (JMS), 409
<body-content> tag, 608
BodyPart objects, 477-478
BodyTag interface, 608
BodyTagSupport class, 609, 622
Book class, 115
Book Manager application, 836-837
BookManagerClient.java, 843
BookManagerClient2.java, 848
BookManagerEJB.java, 841-842
BookManagerEJB2.java, 846-847
CceConnectionSpec class, 838
home interface, 837
IndexedRecord object, 840
InteractionSpec interface, 839
LocalTransaction interface, 845
MappedRecord object, 840
methods
begin(), 845
close(), 841
createInteraction(), 839
execute(), 840, 845
getConnection(), 839
getRecordFactory(), 839
insertBook(), 837
iterator(), 845
listTitles(), 844
lookup(), 838
rollback(), 846
setSessionContext(), 838
remote interface, 837
BookFactory.java file, 118
BookManagerClient.java file, 843
BookManagerClient2.java file, 848
BookManagerEJB.java file, 841-842
BookManagerEJB2.java file, 846-847
BookRef.java file, 117-118
books
Design Patterns - Elements of Reusable Object-Oriented Software, 791
The Timeless Way of Building, 788
UML Distilled, Second Edition, 966
boolean type, 717
**
 tag (HTML), 511**
bulletin board application
publisher, 417-418
subscriber, 418-420
Business Delegate pattern
case study analysis, 820-821
defined, 796
maintainability, 821
performance, 821
reliability, 821
business interface (EJB), 264
implementing, 135-138
methods, 132
patterns, 203
business logic, 12, 134
Entity and Session EJBs (Enterprise JavaBeans), 212-213
separating from presentation tier, 130-131
business-tier patterns
Business Delegate
case study analysis, 820-821
defined, 796
maintainability, 821
performance, 821
reliability, 821
Composite Entity
case study analysis, 812-813
defined, 796
flexibility, 813
maintainability, 813
performance, 813
Fast Lane Reader, 796
Page-by-page Iterator, 796
Session Façade
case study analysis, 798-800
defined, 795
flexibility, 799
maintainability, 799
performance, 799
security, 799
Value List Handler, 796
Value Object
case study analysis, 800-804
defined, 796
Partial Value Object, 804
Value Object Builder, 796
business-to-business (B2B) applications, 129

BusinessLifeCycleManager interface, 936
BusinessQueryManager interface, 937
<BUTTON> tag (HTML), 511
BytesMessage message type, 409

C

caches, 384-387
Caesar cipher, 656-657
calling RPC-style Web Services
ServiceClient class, 891
SOAP (Simple Object Access Protocol), 889-891
cardinality, 281
CAs (Certification Authorities), 660
cascade delete relationships, 312
cascade nulls, 281
<cascade-delete> tag, 317
case sensitivity (XML), 988
case study. *See* Agency case study
CaseStudy directory, 76
catalogs of patterns, 792
CceConnectionSpec class, 838
CCI application, 836-837
BookManagerClient.java, 843
BookManagerClient2.java, 848
BookManagerEJB.java, 841-842
BookManagerEJB2.java, 846-847
CceConnectionSpec class, 838
home interface, 837
IndexedRecord objects, 840
InteractionSpec interface, 839
LocalTransaction interface, 845
MappedRecord objects, 840
methods
begin(), 845
close(), 841
createInteraction(), 839
execute(), 840, 845
getConnection(), 839
getRecordFactory(), 839
insertBook(), 837
iterator(), 845
listTitles(), 844
lookup(), 838
rollback(), 846
setSessionContext(), 838
remote interface, 837
CD-ROM
Agency directory, 76
CaseStudy directory, 76
Examples directory, 76
Exercise directory, 76-77
Solution directory, 76
CDATA attribute type, 713, 994
certificate realm, 664
certificates (digital), 660-661
Certification Authorities (CAs), 660
characters() method, 723
checksums, 660
child axis (XPath), 762
<choose> tag, 645
cipher keys, 657
Class diagrams
associations, 969-970
attributes, 970-971
constraints, 973
generalization, 972
operations, 971-972
classes, 15
abstract, 273
Adapter, 204-205
Authenticator, 494, 499
bean, 431
BodyTagSupport, 609, 622
Book, 115
CceConnectionSpec, 838
custom primary keys, 227-229
dependent value, 259-261
DirContext, 108
Factory, 117
InitialContext, 86
installing as jar files, 374
javax.ejb package, 167-168
JDO (Java Data Objects), 387-389
loading from code bases, 114
MessageSender, 448-449, 736-737
MyAuthenticator, 494-495
PTPReceiver, 413-414
PTPSender, 410-411
RecipientType, 498
ServiceClient, 891
ServiceDefinition, 933
ServiceProvider, 932
ServiceRegistryProxy, 932
servlet class hierarchy, 513
SessionContext, 168
SQLj Part 0, 368
super and sub, 381
TagSupport, 609
TEI (Tag Extra Info), 635-637
UDDIProxy, 931
UnicastRemoteObject, 853
wrapping as Web Services deployment descriptors, 895-896

deployment information, 897-898
SimpleOrderServer.java example, 894-895
XALAN, 748
XSLT (Extensible Stylesheet Transformations), 751-755
CLASSPATH environment variable, 31, 85
clauses (SQL)
FROM, 983
GROUP BY, 984
HAVING, 984
ORDER BY, 984-985
WHERE, 983-984
cleanup tool, 65
client tier
applet clients, 51
dynamic HTML clients, 50-51
mobile devices, 51
non-Java clients, 54
peer-to-peer communication, 53
standalone clients, 52-53
static HTML clients, 49-50
Web Services, 54
client-demarcated transactions, 350
client-side proxies, 820-821
clients, 49
applet clients, 51
authentication, 655
BeanOrderService client, 916-918
BookManagerClient.java, 843
BookManagerClient2.java, 848
dynamic HTML clients, 50-51
EJBs (Enterprise JavaBeans), 126, 129
HelloUserClient.java, 855, 859
JAXM (Java API for XML Messaging), 941
JMS (Java Message Service), 399
message sender clients, 445-449
message-based Web Services, 938-939
JAXMOrderServer.java, 952-954
JAXMOrderService Client.java, 943-944
ProcessingServlet.java, 960-961
running, 947
SubmittingServlet.java, 957-959
Message-driven beans and, 430-431
mobile devices, 51
non-Java clients, 54
peer-to-peer communication, 53
redirecting, 529
remote clients, 127
running EJBs (Enterprise JavaBeans), 150-151
standalone clients, 52-53
static HTML clients, 49-50
thin clients, 9, 13
user credentials, 655
Web Service clients
SimpleOrderClient.java example, 899-900
WSDL (Web Services Description Language) descriptions, 898-899
Web Services, 54
XML (Extensible Markup Language) data, 742-743
CLOB data type, 366, 375
cloneNode() method, 731
close() method, 448, 487, 841
closing
JMS (Java Message Service) connections, 410
RI (Reference Implementation), 37
Cloudscape
Cloudscape Server tool, 65
diagnostic messages, 34
starting, 34
troubleshooting, 34
read-only installation directory, 35
refused connections, 36-37
server port conflicts, 35-36
CMP (container-managed persistence), 217, 271-273
Agency database, 276-277
Entity EJBs (Enterprise JavaBeans)
abstract accessor methods, 274-275
abstract classes, 273
CMR (container-managed relationships), 279
cmr-fields, 282-285
configuring, 313-322
lifecycle management, 277-279
local interface, 301
LocalHome interface, 301
manipulating relationships, 286-291
relationship navigability, 282
relationship types, 280-281

cmp-fields
exposing, 324-325
naming restrictions, 275
CMR (container-managed relationships), 273, 279
cmr-fields, 282-285
exposing, 325-326
naming restrictions, 283
relationships, 282-285
manipulating relationships, 286-291
relationship navigability, 282
relationship types, 280-281
<cmr-field-name> tag, 318
<cmr-field-type> tag, 318
cmr-fields
exposing, 325-326
naming restrictions, 283
relationships, 282-285
code bases
defining, 114-117
Book.java class, 115
JNDICodebase.java, 115-116
JNDILookupBook.java, 116-117
loading classes from, 114
code listings
acquire late, release early database connections, 262
Advertise interface, 801-802
advertise.jsp, 592-594, 632-634, 693
AdvertiseBean.ejbCreate() method, 197
AdvertiseBus interface, 204
AdvertiseJob bean
deployment descriptor, 339
updateDetails() method, 255-256
updateDetails() method BMTD implementation, 347-349
updateDetails() method with Entity bean layer, 256-257
AdvertiseValueObject object, 803
agency.css, 589
agency.jsp, 590, 692
agency.ldif, 105
AgencyBean
AgencyBean.java, 582-584, 817-819
business method implementation, 136-137
createCustomer() method, 681
deleteCustomer() method, 681
deployment descriptor, 144-145
ejbCreate() method, 175
home interface, 140
getLocations() method, 177-178
lifecycle methods, 139
agencyName.jsp, 581
AgencyTable servlet, 550-552
ApplicantMatch bean, 454-456
ApplyXSLT.java, 752-753
<assembly-descriptor> tag, 668
AuditFilter servlet
doFilter() method, 545
source code, 537-538
banner.html, 571
basicHTML.xsl, 757
BeanOrderServer.java, 915
BeanOrderService
client, 916-918
serializer definition, 915
Book.java, 115
BookFactory.java, 118
BookManagerClient.java, 843
BookManagerClient2.java, 848
BookManagerEJB.java, 841-842
BookManagerEJB2.java, 846-847
BookRef.java, 117-118
bulletin board program
publisher, 417-418
subscriber, 418-420
Cloudscape startup diagnostics, 34
CMP Entity EJB abstract accessor methods, 274
color attribute validation, 636-637
createJob() method, 811
date.jsp, 560
dateBanner.jsp, 570
dd.xml, 772
demo.tld, 606
deploy.xml, 902-903
DirectJobDEOImpl implementation, 807-809
DOM Parser, 728-731
dynamic pages
as JSP (JavaServer Page), 557
as servlet, 556
ejbActivate() and ejbPassivate() methods, 235, 304
ejbCreate() method, 305, 679
ejbCreater() and ejbPostCreate() methods, 235, 237
ejbHomeDeleteByCustomer() method, 310

ejbLoad() method, 232-234, 303, 369-370
ejbRemove() method, 238-239, 307
ejbStore() method, 232-234, 303
Entity beans, referencing, 257
<entity> tag, 250, 314-316
errorPage.jsp, 595-596
finder methods, 239-240
footer.jsf, 591
foreach.jcp, 643
ForEachTag.java
  source code, 623-624, 628-629
  TLD (tag library descriptor), 625, 630-631
getCustomerObj() method, 313
GetCustTag.java, 620-621
header.jsf, 588
hello.jsp, 605
HelloServer.java, 887
HelloServerClient.java, 888
HelloServerPortType.java, 886
HelloTag.java, 611
HelloUser.java, 852
HelloUserClient.java, 855, 859
HelloUserImpl.java, 854, 858
HelloWorld.java, 863
HTML (Hypertext Markup Language) form, 511-512
HTML (Hypertext Markup Language) page, 510
HTMLPage with counter, 541
Imp.c, 864
JavaMail
  AuthenticateRetrieve Mail application, 496-497
  RetrieveAttachment application, 492-494
  SendAttachmentMail.java, 483-484
  SendHTMLMail.java, 474-475
  SendMail.java, 470-471, 475
  SendMultiPartMail.java, 480-481
JAXMOrderServer.java, 953-954
JAXMOrderServiceClient.java, 943-944
JAXR client initialization code, 935
JDO (Java Data Objects)
  classes, creating, 389
  criteria-based searching, 390
jndi.properties file, 697
JNDIAttributes.java, 108-109
JNDIBind.java application, 91
JNDIBindBookRef.java, 119
JNDICodeBase.java, 115-116
JNDICreate.java, 99
JNDIDestroy.java, 100
JNDIFilter.java, 111
JNDIListSAMS.java, 96-97
JNDILookup.java, 93-94
JNDILookupAny.java, 107
JNDILookupBook.java, 116-117
JNDILookupBookRef.java, 119-120
JNDILookupSAMS.java, 94-95
JNDIModify.java, 112-113
JNDISearch.java, 110
JNDITree.java, 98
job.xml, 749
Job/Skill relationship, 319
JobBean
  business methods, 242
  ejbHomeDeleteBy Customer() method, 241
  setEntityContext() method, 231-232
JobDAO interface, 806
JobData class Job bean state information, 379
JobLocal interface, 230, 301
JobLocalHome interface, 225, 301
JobPK class, 227-228, 309
jobSummary document
  attributes, 709
  DTD (document type declaration), 714
  initial source code, 708
  namespace, 714-715
  XML Schema, 716-717
JobValueObject object, 806-807
jobs.xml
  applying basicHTML to, 757-758
  applying textHTML.xsl to, 760-761
LineItemBean.java, 914
list() method, 96-97
listBindings() method, 97-98
listcust.jsp, 644
ListSASL.java, 697
loadDetails method, 803-804

Location/Job relationship, 318
<login-config> tag, 685
lookup.jsp, 615
LookupTag.java, 616-617
MDBPrintMessage bean
  deployment descriptor, 444-446, 455
  source code, 440
<method-identity> tag, 671
MyHelloService.wsdl, 883-884
name.jsp, 572-573
NamingListener object, 121
narrow() method, 96
OptionTag.java
  source code, 629-630
  TLD (tag library descriptor), 630-631
OptionTagTEI.java, 636
Order.java, 948-950
point-to-point messaging
  queue receiver, 413-414
  queue sender, 410-411
port conflict error message, 35-36
ProcessingServlet.java, 960-961
RegisterBusiness.java, 930-931
Remote interface for Agency EJB, 133-134
remote interface for stateful Advertise bean, 198
RI (Reference Implementation) startup diagnostics, 33-36, 78-79
role references, 680, 694
<run-as> tag, 676
SAX Parser, 722-723
<security-constraint> tag, 690-691
<security-identity> tag, 670
select.jsp, 638
SelectTag.java, 638-639
Servlets application, 514-515
session.xls, 773
session.xml, applying to dd.xml, 774
SessionSimpleOrderServer.java, 906
  client code, 907-908
  deployment descriptor, 906-907
setEntityContext() and unsetEntityContext() methods, 302-303
simple.xsl, 746
SimpleOrderClient.java, 899-900
SimpleOrderServer.java, 895
SimpleOrderServer2.jws, 903-904
SimpleOrderServerSoapBindingImpl.java file, 902
SimpleOrderServerSoapBindingSkeleton.java file, 901-902
SimpleOrderService
  deployed services, 897-898
  deployment descriptor, 895
simpleSpace.xsl, 768
simpleStrip.xsl, 767
skills.jsp, 627-628
SoapSayHello.java, 889-890
SQL (Standard Query Language)
  transaction fragment, 338
  UDFs, 374
SQLj version of ejbLoad(), 371-372
stateless Agency bean remote interface, 174
static methods as SQL stored procedures, 376
SubmittingServlet.java, 957-959
table.jsp, 576-577
table.xsl, 765
tableForm.jsp, 576
tableStyle.xsl, 776
tag library, 820
textHTML.xsl, 759
updateCustomer.jsp, 595
UserTransaction objects, obtaining, 350
valid XML (Extensible Markup Language) elements, 706
VerifyData servlet
  parameters, 521
  VerifyForm HTML page, 520
Web application deployment descriptor, 526
XLAN from the command line, 750
XML (Extensible Markup Language) example, 705
XSL-FO document, 744
xsltForm.html, 754

ColdFusion, 24
Collection interface, 287
comma (,), 993
Command design pattern, 205, 794
commands. *See also* methods
  deploytool, 405-406
  j2eeadmin, 404-405
  slapadd, 106
  slatcat, 106
comment() function, 763
comments
  JSPs (JavaServer Pages), 560

XML (Extensible Markup Language), 709-710, 769-770, 990
commit() method, 346, 423-424
common client interface (Connector architecture), 834
Common Object Request Broker Architecture. *See* CORBA
Common Object Services (COS), 18
compatibility of servers, 24-25
compilation errors, 567-568
compilers, 780-781
completeTransactionIf Required() method, 349
complex type mapping, 912-919
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
complex types, 717
composing
entities, 812-813
JSPs (Java Server Pages), 813-817
Composite Entity pattern
case study analysis, 812-813
defined, 796
flexibility, 813
maintainability, 813
performance, 813
composite names, 100-101
Composite View pattern
case study analysis, 813-817
admin.jsp, 815
agency.jsp, 814
flexibility, 816
footer.jsf, 815
header.jsf, 814
maintainability, 816
manageability, 816
performance, 817
defined, 795
compound names, 101
Concept interface, 936
conceptual level (UML), 966
confidential option (network security requirements), 690
confidentiality, 655
config object, 575
configuring
Basic authentication, 684-685
EJBs (Enterprise JavaBeans)
BMP (bean-managed persistence), 248-252
CMP (container-managed persistence), 313-322
metadata, 141-142
stateful EJBs, 200
stateless EJBs, 180-181
environment variables
CLASSPATH, 31
JAVA_HOME, 29-30
PATH, 30-31
JavaBean properties, 580-581
JAXM (Java API for XML Messaging), 941-942
JNDI (Java Naming and Directory Interface), 85-88
applet parameters, 90
application properties, 89
CLASSPATH variable, 85
hard-coded properties, 90
J2EE RI for Linux and Unix, 86
J2EE RI for Windows, 85-86
jndi.properties file, 88-89
server startup, 86
transactions
deploytool, 342
XADataSource interface, 357
XALAN, 748-749
conflicts (server port), 35-36
connect() method, 486, 494
ConnectException exception, 36-37
connection contract (Connector architecture), 831
connection factories, 403
Connection interface, 356-358
connection pooling, 357
Connection Refused error message, 36-37
ConnectionFactory object, 403
ConnectionPoolDataSource interface, 357
connections
databases, 177
JMS (Java Message Service), 399
closing, 410
ObjectConnection object, 416
refused connections, 36-37
SQLj, 372
connectivity technologies, 848-849
choosing, 865
CORBA (Common Object Request Broker Architecture), 849-851

IDL (Interface Definition Language), 850
IIOP (Internet Inter-ORB Protocol), 850
Naming Service, 851
ORB (Object Request Broker), 850
J2EE Connector architecture. *See* Connector architecture
Java IDL (Interface Definition Language), 851
JNI (Java Native Interface), 860-865
HelloWorld.java example, 863
HelloWorldImp.c example, 864
when to use, 865
legacy connectivity, 22
RMI (Remote Method Invocation), 851-852
RMI over IIOP, 857-860
RMI over JRMP, 852-857
when to use, 865
Connector architecture, 62, 829-830
CCI application, 836-837
BookManagerClient.java, 843
BookManagerClient2.java, 848
BookManagerEJB.java, 841-842
BookManagerEJB2.java, 846-847
CceConnectionSpec class, 838
home interface, 837
IndexedRecord objects, 840
InteractionSpec interface, 839
MappedRecord objects, 840
methods, 837-841, 844-846
remote interface, 837
common client interface, 834
connection contract, 831
EIS (Enterprise Information Systems) interaction, 834-835
resource adapter installation, 835-836
roles, 830-831
security contract, 833-834
transaction management, 843
begin() method, 845
BookManagerClient2.java example, 848
BookManagerEJB2.java example, 846-847
contract, 832-833
execute() method, 845
iterator() method, 845
listTitles() method, 844
LocalTransaction interface, 845
rollback() method, 846
Web site, 831
when to use, 865
constants, Status interface, 346, 360
constraints, 686, 973
creating, 687-689
<security-constraint> tag, 690-691
consuming messages
JMS (Java Message Service), 411-412
Message-driven beans, 435-436
container-managed persistence. *See* CMP
container-managed relationships. *See* CMR
container-managed transaction demarcation, 338-344, 352, 392, 436-437
<container-transaction> tag, 340
containers, 20-21, 55-56. *See also* CMP (container-managed persistence); CMR (container-managed relationships)
EJBs (Enterprise JavaBeans)
deploying, 147-148
EJB 2.0 specification, 172performance tuning, 247-248
services, 126, 131-132
servlet containers, 513
content types
e-mail, 469, 479
HTTP (Hypertext Transfer Protocol) responses, 508
Content-Disposition header (MIME), 490, 498
Context object. *See* contexts
contextDestroyed() method, 542
contextInitialized() method, 542
contexts
changing, 94-95
creating, 98-99
destroying
destroySubcontext() method, 99
JNDIDestroy.java application, 99-100
directory contexts, 108

initial contexts
creating, 86
naming exceptions, 86-87
listing
JNDIListSAMS.java example, 96-97
JNDITree.java example, 98
list() method, 96-97
listBindings() method, 97-98
Message-driven beans, 433
naming exceptions, 86-87
ServletContext interface, 524
contracts (Connector architecture)
connection, 831
security, 833-834
transaction management, 832-833
cookies, 532-533
creating, 533-534
retrieving information from, 534
cooperating tags
hierarchical tag structures
advertise.jsp page, 632-634
findAncestorWithClass() method, 627
<forEach> tag, 628-630
getParent() method, 627
<option> tag, 629-631
skills.jsp page, 627-628
shared script variables, 626
copyrights, 29
CORBA (Common Object Request Broker Architecture), 849-851
COS (Common Object Services), 18
IDL (Interface Definition Language), 850
IIOP (Internet Inter-ORB Protocol), 850
Naming Service, 851
ORB (Object Request Broker), 850
COS (Common Object Services), 18
CREATE TABLE statement (SQL), 979
CREATE VIEW statement (SQL), 979-980
create() method, 140, 173
CreateAgency.bat file, 77
CreateAgency.class file, 76
CreateAgency.java file, 76
CreateAgency.sh file, 77
createAttachment() method, 951
createCustomer() method, 681
createDurableSubscriber() method, 420-421
createInteraction() method, 839
createJob() method, 810-811
createReceiver() method, 412
createSubcontext() method, 99
createSubscriber() method, 420-422
createTextMessage() method, 409
credentials, 655
credentials property (JNDI), 122, 695
CROSS JOIN statement (SQL), 982
cross joins, 982
cryptography
Caesar cipher, 656-657
JCE (Java Cryptography Extension), 698
CSS (Cascading Stylesheets), 742
custom tags
attributes
lookup tag example, 615-617
request time expressions, 616
TLDs (tag library descriptors), 616
validating, 635-637
example, 605-606, 611-612
hierarchical tag structures
advertise.jsp page, 632-634
findAncestorWithClass() method, 627
<forEach> tag, 628-630
getParent() method, 627
<option> tag, 629-631
skills.jsp page, 627-628
interfaces, 608
iterative tags, 622
body content, 622-623
BodyTagSupport class, 622
<forEachJob> tag example, 623-626
IterationTag interface, 622
lifecycle
doAfterBody() method, 610-611
doEndTag() method, 610
doInitBody() method, 611
doStartTag() method, 610
release() method, 610
script variables, 618
adding to page contexts, 618
defining, 637

<getCust> tag example, 619-622
sharing, 626
TLDs (tag library descriptors), 618-619
support classes, 609
tag body processing
doSelect() method, 640
<select> tag example, 637-639
TLDs (tag library descriptors)
attributes, 616
creating, 606-608
example, 606
file location, 614
<forEach> tag, 630
script variables, 618-619

D

data access, 22
data access logic, 12
Data Access Object pattern, 804-806
createJob method, 810-811
DirectJobDAOImpl implentation, 807-809
JobDAO interface, 806
JobValueObject object, 806-807
EJB services, 131
servlet variables, 575
UDDI (Universal Description, Discovery, and Interaction)
JAXR (Java API for XML Registries), 934-937
locally hosted registries, 929
public production registries, 929
public test registries, 929
UDDI4J, 929-932
WSKT Client API, 932-934
Data Access Object pattern
case study analysis, 804-811
createJob method, DirectJobDAOImpl implementation, 807-809
JobDAO interface, 806
JobValueObject object, 806-807
defined, 796
flexibility, 805, 811
maintainability, 805, 811
performance, 811
data exchange. *See* Value Object pattern
data integrity
checksums, 660
defined, 655-656
message digests, 660
data store transactions, 385
data types, 717
SQL1999, 366
type mapping
serializers, 912-919
SOAP/WSDL types, 911-912
databases
Agency
CMP support, 276-277
installing, 77-78
Cloudscape
diagnostic messages, 34
starting, 34
troubleshooting, 34-37
Entity EJBs (Enterprise JavaBeans), 214-215, 262-263
JDBC (Java Database Connectivity), 57
joins
cross joins, 982
full joins, 983
inner joins, 981
left outer joins, 982
right outer joins, 982
manipulating, 177
object modeling, 215
RDBMS technology, 213
SQLj, 372
tables
adding rows to, 980-981
creating, 979
deleting rows from, 980
dropping, 980
editing, 978-979
retrieving data from, 981
updating, 229, 983
views
creating, 979-980
dropping, 980
DataSource object
connection pooling, 357
container-managed persistence, 217
date type, 717
date.jsp file, 560-563
date/time information, generating, 560-563
dateBanner.jsp page, 570
debugging RPC-style Web Services, 892-894
declarations
JSPs (JavaServer Pages), 559
XML (Extensible Markup Language), 706, 989-994
declarative authorization
network security requirements, 689-690
roles, 685
security constraints, 686-691

declarative security, 661
<declare> tag, 619
declaring. *See* defining
Decorator pattern, 794
default realm, 664
DefaultHandler methods, 723-724
defining
attributes, 616, 712
code bases, 114-117
Book.java class, 115
JNDICodebase.java, 115-116
JNDILookupBook.java, 116-117
elements, 711-712
environment variables
CLASSPATH, 31
JAVA_HOME, 29-30
PATH, 30-31
JNDI (Java Naming and Directory Interface), 87-88
applet parameters, 90
application properties, 89
hard-coded properties, 90
jndi.properties file, 88-89
roles, 666-668
script variables, 637
XML (Extensible Markup Language) attributes, 993-994
XML (Extensible Markup Language) elements, 992-993
delegates, 820-821
DELETE statement (SQL), 507, 980
deleteByApplicant() method, 452
deleteCustomer() method, 681
DELETED flag (e-mail), 489
deleting
e-mail messages, 489-490
table rows, 980
dependent value classes, 259-261
deploy.xml file, 902-903
deployable components, 143
deployment. *See also* deployment descriptors
bean-managed transaction demarcation, 349
deployable components, 143
EARs (Enterprise Application Archives), 67
EJB-JAR files, 69
EJBs (Enterprise JavaBeans)
containers, 147-148
J2EE RI, 151
stateful EJBs, 200
vendor options, 181
enterprise applications, 193, 322-323
filters, 538, 540-541
JSPs (JavaServer Pages), 597-600
listeners, 543, 545
Message-driven beans, 442-445, 456-457
servlets, 515-518
Tag Libraries, 612, 614
WAR (Web Archive) files, 70
deployment descriptors, 67-68
EJBs (Enterprise JavaBeans), 142-146
CMP (container-managed persistence) EJBs, 313-322
configuring, 180
EJB references, 188-189
Entity EJBs, 248-251
environment entries, 187-188
resource environment references, 192-193
resource references, 190-192
vendor-specific, 186
XML (Extensible Markup Language) documents, 182-186
enterprise applications, 146-147
JDO (Java Data Objects), 391
limitations, 321
transactions, 339, 343
Web applications, 526-527
deploytool, 405-406, 664
deployment settings, 677-678
EJB (Enterprise JavaBeans), 152, 181-182
agency application, 154-157
CMP (container-managed persistence) EJBs, 316
Entity EJBs, 251-252
limitations, 181
method permissions, 670-674
role mappings, 674-676
security identity, 669-670
transaction configuration, 342
XADataSource interface, 357
descendent axis (XPath), 763
descendent-or-self axis (XPath), 763

descriptors, 67-68
deployment. *See* deployment descriptors
TLDs (tag library descriptors)
attributes, 616
creating, 606-608
example, 606
file location, 614
forEach tag, 630
option tag, 630
script variables, 618-619
Web applications, 526-527
design patterns. See patterns *Design Patterns - Elements of Reusable Object-Oriented Software*, 791
destinations
J2EE RI, 404
JBoss, 403-404
destroy() method, 523, 537
destroying contexts
destroySubcontext() method, 99
JNDIDestroy.java application, 99-100
destroySubcontext() method, 99
DeveloperWorks Web site, 792
development, 9-10
Enterprise Computing Model, 17-18
lifecycle, 18
naming, 18-19
persistence, 18
security, 19-20
transactions, 19
monolithic development
disadvantages, 10-11
structure, 10
transitioning to *n*-tier, 26
n-tier design, 13-14, 28, 38
advantages, 16-17
business tier, 39-44
client tier, 49-54
component frameworks, 15-16
modularity, 14-16
presentation tier, 44-49
transitioning to, 26
two-tier design
disadvantages, 12-13
layers, 11-12
diagnostic messages
Cloudscape, 34
RI (Reference Implementation), 33-34, 78-79
diagrams
Class diagrams
associations, 969-970
attributes, 970-971
constraints, 973
generalization, 972
operations, 971-972
Sequence diagrams
activations, 973
example, 974-975
lifelines, 973
messages, 973
Use Case diagrams, 967-969
actors, 967
<<extend>> notation, 968
generalization notation, 968
<<include>> notation, 967
notation, 967
dialog boxes
Environment Variable, 30
New System Variable, 30
Digest authentication, 683
Digest MD5 authentication, 696
digests, 660
digital certificates, 660-661
digital signatures, 658
DirContext class, 108
directives
defined, 558
include, 570-571
page, 571-575
syntax, 570
DirectJobDAOImpl implentation, 807-809
directories. *See also* directory services
Active Directory, 19
Agency, 76
bin, 30
CaseStudy, 76
Examples, 76
Exercise, 76-77
JNDI (Java Naming and Directory Interface), 59
read-only installation directory, 35
Solution, 76
Web applications, 525-526
directory services, 22. *See also* JNDI (Java Naming and Directory Interface)
defined, 82
LDAP (Lightweight Directory Access Protocol), 84
attributes, 102, 108-109, 112-114
obtaining, 103
OpenLDAP, 104-106
Service Providers, 106-107
testing, 107-108
X.500 names, 102-103
NDS (Novell Directory Services), 83
discovery, 148-149
Dispatcher View pattern, 795

<display-name> tag, 183, 527
displayHelloWorld() method, 864
disposal of EJBs (Enterprise JavaBeans), 150
distribution via proxies, 131
DNS (Domain Name System), 19, 83
doAfterBody() method, 610-611, 622-624
<!DOCTYPE> tag, 527, 711, 992
Document interface
appendChild() method, 731
cloneNode() method, 731
getAttributes() method, 726-728
getChildNodes() method, 727
getDocumentElement() method, 727
getElementsByTagName() method, 727
getFirstChild() method, 727
getLastChild() method, 727
getNamespaceURI() method, 726-728
getNodeName() method, 726-728
getNodeType() method, 726-728
getNodeValue() method, 726-728
getParentNode() method, 727
getPreviousSibling() method, 727
hasAttributes() method, 728
hasChildNodes() method, 728
normalize() method, 726
removeChild() method, 731
Document Object Model parsers. *See* DOM parsers
document roots, 525
document type declarations. *See* DTDs
Document Type Definitions. *See* DTDs
documentation
J2EE SDK (Software Developers Kit), 20, 32
patterns, 789
SQL (Standard Query Language), 977
DocumentBuilderFactory interface, 725
documents (HTML)
compared to XML, 705
HTML form example, 511-512
simple HTML page example, 510
VerifyForm page, 520
documents (XML), 701-702, 987-988. *See also* tags
advantages, 703
attributes, 708-709, 712-713
case sensitivity, 988
comments, 709-710, 990
compared to HTML, 705
declarations, 706
defined, 988
deployment descriptors, 67-68
DTDs (document type declarations), 710-711, 989-990
attributes, 712-713
defined, 706
element content, 712
element type declarations, 711-712
example, 713-714
DTDs (Document Type Definitions), 992-995
attribute declarations, 993-994
deployment descriptors, 182, 249
element declarations, 992-993
EJB (Enterprise JavaBeans) references, 188-189
entity references, 994-995
environment entries, 187-188
example, 992
resource environment references, 192-193
resource references, 190-192
presentation elements, 183-184
Session element, 184-186
enforcing structure of, 991
DTDs (Document Type Definitions), 992-995
XML Schemas, 995-997
history of, 703-704
JASB (Java Architecture for XML Binding), 732-733
JAXM (Java APIs for XML Messaging), 25
JAXP (Java API for XML Parsing), 58-59, 718-720
JAXR (Java APIs for XML Registries), 25
jobSummary document
attributes, 708-709
code listing, 708
DTD (document type declaration), 713
namespace, 714-715
XML Schema, 716-717

namespaces, 714-715, 991
online documentation, 997
parsing with DOM (Document Object Model)
  accessing tree nodes, 726-728
  Document interface methods, 725-728, 731-732
  DocumentBuilder Factory interface, 725
  DOM Parser application, 728-731
  modifying tree nodes, 731-732
  parse() method, 725
parsing with SAX (Simple API for XML), 719-720
  DefaultHandler methods, 723-724
  endElement() method, 721
  SAX Parser application, 721-723
  SAXParseFactory interface, 720
  startElement() method, 720-721
platform-independent data exchange, 702-703
prologs, 990
root elements, 705-706
special characters, 990-991
support for, 22
tags
  <assembly-descriptor>, 340, 668
  <attribute>, 616
  attributes, 615-617, 635-637
  <body-content>, 608
  <cascade-delete>, 317
  <choose>, 645
  <cmr-field-name>, 318
  <cmr-field-type>, 318
  content, 712
  custom tags, 608-609
  <declare>, 619
  declaring, 992-993
  <display-name>, 527
  <!DOCTYPE> tag, 527, 711, 992
  element type declarations, 711-712
  <error-page>, 527-528
  example, 605-606, 611-612
  <forEach>, 628-630, 643-644
  <forEachJob>, 623-626
  <forTokens>, 645
  <getCust>, 619-622
  <hello>, 605-606, 611-612
  hierarchical tag structures, 627-634
  <if>, 645
  <init-param>, 527
  iterative tags, 622-626
  <jobSummary>, 708
  lifecycle, 610-611
  <login-config>, 685
  <lookup>, 615-617
  <method-intf>, 671
  <method-permissions>, 670-671
  <name-from-attribute>, 618
  <name-given>, 618
  nesting, 707
  <option>, 629-631
  <param-name>, 527
  <param-value>, 527
  <run-as>, 676
  scope, 619
  script variables, 618-622, 626
  <security-constraint>, 690-691
  <security-identity>, 670
  <servlet-class>, 527
  <servlet-mapping>, 527
  <servlet-name>, 527
  <session-config>, 527
  special characters, 707
  structure of, 988-989
  <tag>, 608
  tag body processing, 637-640
  <taglib>, 607
  TLDs (tag library descriptors), 606-608, 614-619, 630
  <transaction-type>, 437
  <variable>, 618-619
  <variable-class>, 619
  valid elements, 706
  XSLT, 779-780
  <web-app>, 527
transformations, 742, 746-747
  adding comments, 769-770
  attribute values, 770-771
  compilers, 780-781
  creating elements, 771-774
  defining attributes, bv774-776
  elements, 779-780
  numbering elements, 777-778
  whitespace, 767-769
valid documents, 704
validating, 705
well-formed documents, 704, 708
XML Schemas, 715-716, 995-997
  Agency case study, 734
  example, 716-717

schema type definitions, 717-718
validator, 716
XPath, 997-998
XPointer, 997
doEndTag() method, 610
Does Not Exist state (Message-driven beans), 432
doFilter() method, 535-536, 544-545
doGet() method, 522, 959
doInitBody() method, 611
DOM (Document Object Model) parsers
Document interface methods
appendChild(), 731
cloneNode(), 731
getAttributes(), 726-728
getChildNodes(), 727
getDocument Element(), 727
getElementsByTag Name(), 727
getFirstChild(), 727
getLastChild(), 727
getNamespaceURI(), 726-728
getNodeName(), 726-728
getNodeType(), 726-728
getNodeValue(), 726-728
getParentNode(), 727
getPreviousSibling(), 727
hasAttributes(), 728
hasChildNodes(), 728
normalize(), 726
removeChild(), 731
DocumentBuilderFactory interface, 725
DOM Parser application, 728-731
parse() method, 725
Domain Name System (DNS), 19, 83
domains
DNS (Domain Name System), 19, 83
JMS (Java Message Service)
defined, 399
point-to-point, 400-402
publish/subscribe, 401, 415-416
doPost() method, 522, 954
doSelect() method, 640
doStartTag() method, 610, 621, 624
double quote (“), 101, 990
downloading
J2EE SDK (Software Developers Kit), 30
JSPTL (JavaServer Pages Standard Tag Library), 641
LDAP (Lightweight Directory Access Protocol), 103
DRAFT flag (e-mail), 489
DROP TABLE statement (SQL), 980
DROP VIEW statement (SQL), 980
dropping tables/views, 980
DTDs (document type declarations), 710-711, 989-990
attributes
attribute types, 712-713
default values, 713
defining, 712
defined, 706
element content, 712
element type declarations, 711-712
example, 713-714
DTDs (Document Type Definitions), 992-995
attribute declarations, 993-994
deployment descriptors, 182, 249
EJB (Enterprise JavaBeans) references, 188-189
environment entries, 187-188
presentation elements, 183-184
resource environment references, 192-193
resource references, 190-192
Session element, 184-186
element declarations, 992-993
entity references, 994-995
example, 992
DuplicateKeyException exception, 306
DUPS_OK_ ACKNOWLEDGE message, 423, 438
durable subscriptions, 420-421

E

EAI (Enterprise Application Integration) applications, 129
EARs (Enterprise Application Archives), 67
ebXML (Electronic Business XML), 873

ebXML R&R (Registry and Repository), 926-927
e-commerce applications, 129
editing
attributes
JNDIModify.java application, 112-114
ModificationItem objects, 112
ModifyAttributes() method, 112
tables, 978-979
efficiency of servlets, 503
EIS (Enterprise Information Systems)
CORBA (Common Object Request Broker Architecture), 849-851
IDL (Interface Definition Language), 850
IIOP (Internet Inter-ORB Protocol), 850
Naming Service, 851
ORB (Object Request Broker), 850
guidelines, 865
J2EE Connector architecture, 829-830
CCI application, 836-843
common client interface, 834
connection contract, 831
EIS interaction, 834-835
resource adapter installation, 835-836
roles, 830-831
security contract, 833-834
transaction management, 843-848
transaction management contract, 832-833
Web site, 831
when to uses, 865
Java IDL (Interface Definition Language), 851
JNI (Java Native Interface), 860-865
HelloWorld.java example, 863
HelloWorldImp.c example, 864
when to use, 865
RMI (Remote Method Invocation), 851-852
RMI over IIOP, 857-860
RMI over JRMP, 852-857
when to use, 865
EJB Observer pattern, 796
EJB QL
from clause, 293-295
functions, 299
input parameters, 298
OO/relational impedance mismatch, 300
operators, 297-299
select clause, 291-297
syntax, 293
where clause, 297-300
wildcards, 298
EJB servers, 126
<ejb-class> tag, 145, 185
ejb-jar 2 0.dtd file, 182
<ejb-jar> tag, 182, 248
EJB-JAR files, 69
CMR (container-managed relationships), 280
deployment descriptors, 146
<ejb-name> tag, 145
<ejb-name> tag, 185
<ejb-relation> tag, 317
<ejb-relationship-role> tag, 317
ejbActivate() method, 172, 198, 221, 223, 279, 304
EJBContext interface, 168, 343
ejbCreate() method, 170, 173, 176, 197, 222, 235-237, 278, 305, 433-434, 439, 451, 679
EJBException exception, 180
ejbFindByPrimaryKey() method, 224-226, 240
EJBHome interface, 167
ejbHomeDeleteBy Customer() method, 303
EJBHomeHandle interface, 168
ejbLoad() method, 221-223, 232, 279, 303
EJBLocalHome interface, 167, 218, 225-229
EJBLocalObject interface, 167
EJBObject interface, 132, 135, 167
ejbPassivate() method, 172, 198, 221-223, 279, 304
ejbPostCreate() method, 221-222, 235-237
ejbRemove() method, 171, 177, 223, 279, 307, 435, 439, 451
EJBs (Enterprise JavaBeans), 21, 40-43, 126. *See also* Entity EJBs; Session EJBs
Adapter classes, 204-205
Agency case study
deploying, 156-157
remote interface, 133
testing, 158-160
troubleshooting, 160-161

applications deployment, 193
application development modeling, 127
ApplicationMatch example
  code listing, 453-456
  deleteByApplicant() method, 452
  ejbCreate() method, 451
  ejbRemove() method, 451
  findByLocation() method, 452
  getLocation() method, 452
  getSkills() method, 453
  InitialContext interface, 451
  onMessage() method, 452
  skillMatch counter, 453
bean-managed transaction demarcation
  deployment, 349
  restrictions, 345
  Session EJBs, 345-349
BookManager application
  BookManagerClient.java, 843
  BookManagerClient2.java, 848
  BookManagerEJB.java, 841-842
  BookManagerEJB2.java, 846-847
business interface
  implementing, 135-138
  methods, 132
  patterns, 203
business logic, 130-131, 134
classes, 128
clients
  client-demarcated transactions, 350
  types of, 129
common uses, 128-129
configuration information, 141-142
containers, 126
  container-managed transaction demarcation, 338-344
  performance tuning, 247-248
  services, 126, 131-132
creation cycle, 142-143
creation restrictions, 143
declarative security
  method permissions, 670-674
  role mappings, 674-676
  roles, 666-668
  security identity, 668-670, 676-678
deploying. *See also* deploytool
  containers, 147-148
  deployable components, 143
  J2EE RI, 151
  vendor options, 181
deployment descriptors, 142-146, 180
  application component relationships, 146-147
  EJB references, 188-189
  environment entries, 187-188
  resource environment references, 192-193
  resource references, 190-192
  vendor-specific, 186
  XML document presentation elements, 183-184
  XML document Session element, 184-186
  XML documents, 182-183
EJB 2.0 specification, 143, 172
EJB QL
  from clause, 293-295
  functions, 299
  input parameters, 298
  OO/relational impedance mismatch, 300
  operators, 297-299
  select clause, 291-297
  syntax, 293
  where clause, 297-300
  wildcards, 298
external resource names, 145
factories, 140
file I/O (input/output), 143
HOME environment variable, 152
home interface, 140-141
implementing
  discovery step, 148-149
  disposal step, 150
  retrieval step, 149
  running, 150-151
J2EE Blueprints, 127
lifecycle management, 138-140
Message-driven beans, 429-430
  Agency case study, 447-457, 735-736
  alternative architectures, 457-458
  ApplicantMatch bean, 737-739
  clients, 430-431
  compared to Session and Entity beans, 431
  context, 433
  creating, 434, 439-440

defined, 430
deploying, 442-445, 456-457
exception handling, 436
interface implementation, 439-440
interfaces, 431-432
life cycle, 432-433
MDBPrintMessage bean, 440, 443-445
message acknowledgements, 437-438
message handling, 435-436
message selectors, 438
MessageSender class, 736-737
method-ready pool, 434-435
queues, 441, 456
removing, 435
sender clients, 445-449
state, 431-432
testing, 457
transactions, 436-437
methods
exceptions, 138
invoking, 169
restrictions, 143
programmatic security, 666
Agency case study, 679-682
getCallerPrincipal() method, 678
isCallerInRole() method, 678
programming advantages, 130
servers, 126
services, 205
static member variables, 143
threads, 143
transactions
exceptions, 350-351
stateful Session beans, 352-353
types, 128
usefulness, 129
ejbStore() method, 221-223, 232, 279, 303
Electronic Business XML. *See* ebXML
<!ELEMENT> tag, 992
elements (XML). *See* tags
e-mail (JavaMail), 22, 461-465
addressing, 473
attachments
creating, 482-483
retrieving, 490-494
sending, 483-485
deleting, 489-490
development environment, 465-466
HTML e-mail
absolute URLs, 475
addressing, 473
body text, 473
java.io package, 472
sending, 473-475
IMAP (Internet Message Access Protocol), 464
mail sessions, 468
message flags, 489
MIME (Multipurpose Internet Mail Extensions), 464-465
multi-part messages, 476
body text, 477
BodyPart objects, 477-478
content type, 479
creating, 476-479
headers, 479
images, 479
MimeMultipart objects, 478-479
sending, 479-482
NNTP (Network News Transport Protocol), 464
plain text e-mail, 466
addressing, 469
body text, 469
content type, 469
mail environment properties, 467-468
main() method, 467
MimeMessage objects, 468
sending, 470-471
SMTP host access, 467
subjects, 469
POP3 (Post Office Protocol), 463-464
retrieving, 485
attachments, 490-494
close() method, 487
connect() method, 486
getFolder() method, 486
getMessage() method, 486
getMessages() method, 486
getStore() method, 486
open() method, 486
RetrieveMail application, 487-488
writeTo() method, 487
SMTP (Simple Mail Transfer Protocol), 463
user authentication
AuthenticateRetrieveMail application, 495-497
Authenticator class, 494
MyAuthenticator class, 494-495
Password Authentication object, 495

EMPTY content type (XML), 993
Empty value (<body-content> tag), 608
encodeUrl() method, 535
encrytion
asymmetric, 658-659
symmetric
algorithms, 657-658
Caesar cipher, 656-657
endDocument() method, 723
endElement() method, 721, 723
endPrefixMapping() method, 723
enforcing XML (Extensible Markup Language) document structure, 991
DTDs (Document Type Definitions), 992-995
attribute declarations, 993-994
element declarations, 992-993
entity references, 994-995
example, 992
XML Schemas, 995-997
Entensible Stylesheet Language. *See* XSL
Enterprise Application Archives (EARs), 67
Enterprise Application Integration (EAI), 129
enterprise applications
component relationship descriptions, 146-147
deploying, 193, 322-323
Enterprise Computing Model, 17-18
lifecycle, 18
naming, 18-19
persistence, 18
security, 19-20
transactions, 19
Enterprise Information Systems. *See* EIS
Enterprise JavaBeans. *See* EJBs
enterprise program development, 9-10
Enterprise Computing Model, 17-18
lifecycle, 18
naming, 18-19
persistence, 18
security, 19-20
transactions, 19
monolithic development
disadvantages, 10-11
structure, 10
transitioning to *n*-tier, 26
n-tier design, 13-14, 28, 38
advantages, 16-17
business tier, 39-44
client tier, 49-54
component frameworks, 15-16
modularity, 14-16
presentation tier, 44-49
transitioning to, 26
two-tier design
disadvantages, 12-13
layers, 11-12
<enterprise-beans> tag, 183, 340
ENTITIES attribute type, 713, 994
ENTITY attribute type, 713, 994
Entity EJBs (Enterprise Java Beans), 43-44, 128, 211-212
accessing, 258-259
BMP (bean-managed persistence)
configuring, 248-252
defining interfaces, 225-230
implementing, 231-248
business interface, 264
business logic, 212-213
checklist, 264
CMP (container-managed persistence), 271-273
abstract accessor methods, 274-275
abstract classes, 273
checklist, 328
CMR (container-managed relationships), 279
cmr-fields, 282-285
configuring, 313-322
defining local interface, 301
defining LocalHome interface, 301
lifecycle management, 277-279
manipulating relationships, 286-291
relationship navigability, 282
relationship types, 280-281
compared to Message-driven beans, 431
compared to RDBMS technology, 213
composite primary keys, 284
creating, 222
databases
connections, 262-263
updating, 229
deployment descriptors, 248-251
entity element, 313-316
relationships element, 317-322
encapsulating fields, 261-262

Entity relationship diagram (ERD), 214
Façade pattern, 258
finder methods, 224, 239-240, 245-247
granularity, 245
identifying, 214-215
javax.ejb package, 216-217
local interfaces, 217-219
passivation, 223-235
primary keys, 214, 222, 227
references, 252
SQLj support, 382-383
state persistence, 259-261
surrogate keys, 243-245
system exceptions, 254
types, 217
updating, 223
<entity> tag, 249, 313-316
entity references (XML), 994-995
EntityBean interface
implementing, 220, 231-235, 302-305
lifecycle management, 219-224
enumeration return type, 262
<env-entry> tag, 145
environment entries (XML EJB deployment descriptors), 187-188
Environment Variable dialog box, 30
environment variables
CLASSPATH, 31, 85
J2EE HOME, 152
JAVA_HOME, 29-30
PATH, 30-31
equals() method, 228
EquiJoins, 981
ERDs (Entity relationship diagrams), 214
error handling. ***See also*** **exceptions**
Agency case study, 595-597
Connection Refused error message, 36-37
exceptions
application, 179
asynchronous messaging, 415
bean methods, 138
ConnectException, 36-37
DuplicateKeyException, 306
EJBException, 180
Entity EJBs (Enterprise JavaBeans), 226
initial content naming exceptions, 86-87
Message-driven beans, 436
NoPermission Exception, 112
NoSuchEntity Exception, 234
NoSuchObjectLocal Exception, 253
RemoteException, 134, 180, 853
servlets, 529-530
Session EJB (Enterprise JavaBeans), 201-202
stateless EJBs (Enterprise JavaBeans), 174, 179-180
system, 179, 254
transactions, 350-351
JSPs (JavaServer Pages), 563-565
compilation errors, 567-568
HTML presentation errors, 568-569
translation errors, 565-567
servlets
HTTP errors, 528-529
servlet exceptions, 529-530
error() method, 180, 723
error page (Agency case study), 595-597
<error-page> tag, 527-528
errorPage directive, 572
errorPage.jsp page, 595-597
event handling
event listeners
deploying, 543-545
HttpSessionActivation Listener, 542
HttpSessionAttribute Listener, 542
HttpSessionListener, 542
NamespaceChange Listener, 120
ObjectChangeListener, 120
ServletContext AttributeListener, 542
ServletContextListener, 542
JNDI (Java Naming and Directory Interface)
addNamingListener() method, 120
NamespaceChange Listener event listener, 120
NamingEvent object, 120
NamingListener object, 120-121
ObjectChangeListener event listener, 120

examineHeaderElements method, 951
Examples directory, 76
exceptions
application, 179
asynchronous messaging, 415
bean methods, 138
ConnectException, 36-37
DuplicateKeyException, 306
EJBException, 180
Entity EJBs (Enterprise JavaBeans), 226
initial content naming exceptions, 86-87
Message-driven beans, 436
NoPermissionException, 112
NoSuchEntityException, 234
NoSuchObjectLocalException, 253
RemoteException, 134, 180, 853
servlets, 529-530
Session EJB (Enterprise JavaBeans), 201-202
stateless EJBs (Enterprise JavaBeans), 174, 179-180
system, 179, 254
transactions, 350-351
execute() method, 840, 845
executing
JAXM (Java API for XML Messaging) clients, 947
JSPs (JavaServer Pages), 557
Executive Committees (JCP), 1001
Exercise directory, 76-77
Expert Groups (JCP), 1000
exporting EJB (Enterprise JavaBeans) services, 127
expressions, 559, 763
<<extend>> notation (Use Cases), 968
Extensible Markup Language. *See* XML
external authentication, 122, 696
external resources, 828-829
Connector architecture, 829-830
CCI application, 836-843
common client interface, 834
connection contract, 831
EIS (Enterprise Information Systems) interaction, 834-835
resource adapter installation, 835-836
roles, 830-831
security contract, 833-834
transaction management, 832-833, 843-848
Web site, 831
when to use, 865
CORBA (Common Object Request Broker Architecture), 849-851
IDL (Interface Definition Language), 850
IIOP (Internet Inter-ORB Protocol), 850
Naming Service, 851
ORB (Object Request Broker), 850
guidelines, 865
Java IDL (Interface Definition Language), 851
JNI (Java Native Interface), 860-865
HelloWorld.java example, 863
HelloWorldImp.c example, 864
when to use, 865
names, 145
RMI (Remote Method Invocation), 851-852
RMI over IIOP, 857-860
RMI over JRMP, 852-857
when to use, 865

F

Façade pattern, 258, 794
factories
BookFactory.java, 118
Factory class, 117
ObjectConnectionFactory object, 416
QueueConnection object, 402
QueueConnectionFactory object, 402
RMIConnectionFactory, 403
RMIXAConnection Factory, 403
UILConnectionFactory, 403
UILXAConnection Factory, 403
XAConnectionFactory, 403
Factory class, 117
Fast Lane Reader pattern, 796
FatalError() method, 723
fields
hidden form fields, 532
JMS (Java Message Service) headers, 408

file I/O (input/output), 143
files. *See also* code listings
EARs (Enterprise Application Archives), 67
EJB-JAR, 69
CMR (container-managed relationships), 280
deployment descriptors, 146
ejb-jar 2 0.dtd, 182
JWS (Java Web Service) files
generated WSDL, 904-905
SimpleOrderServer2.jws example, 903-904
transaction logs, 337
filesystems, NTFS, 337
filters
AuditFilter example
code listing, 537-538
doFilter() method, 544-545
deploying, 538-541
filter chains, 536
JNDI (Java Naming and Directory Interface), 111
methods, 535-537
findAncestorWithClass() method, 627
findByLocation() method, 452
findByPrimaryKey() method, 226, 304
finder methods
Entity EJBs (Enterprise JavaBeans), 224, 239-240, 272, 277
BMP (bean-managed persistence) beans, 226
dangers, 245-247
EJB QL, 291-293
LocalHome interface, 308-309
finding
objects
JNDIFilter.java application, 111
JNDISearch.java application, 110
search() method, 109-111
patterns, 792
services, 821-822
#FIXED value (DTD attributes), 713, 994
FLAGGED flag (e-mail), 489
flat transactions, 337
flexibility
Composite View pattern, 816
Data Access Object pattern, 805
Session Façade pattern, 799
Value Object pattern, 804
View Helper pattern, 820
float type, 717
<FONT> tag (HTML), 511
footer.jsf file, 815
footers (Agency Web site), 591-592
forces, 789
<forEach> tag, 643-644
code listing, 628-629
TLDs (tag library descriptors), 630
<forEachJob> tag, 623-626
<FORM> tag (HTML), 511
formatting XML (Extensible Markup Language) documents
applying stylesheets, 746
browser support, 747
on servers, 747
storing transformed data, 746-747
XSL-FO, 744-745
XSLT (XSL Transformations), 745-746
template rules, 756-761
Transformer class, 751-755
forms
hidden form fields, 532
HTML (Hypertext Markup Language), 511-512
forms-based authentication, 683
<forTokens >tag, 645
forward slash (/), 100-101
FROM clause, 293-295, 983
Front Controller pattern, 795
FULL JOIN statement (SQL), 983
full joins, 983
function aliases, 375
functions. *See also* methods
EJB QL, 299
XPath, 763
future
of J2EE, 25
of patterns, 823

G

"Gang of Four" book, 791
generalization (Class diagrams), 972
generalization notation (Use Cases), 968
generic patterns
Command, 794
Decorator, 794
Façade, 794
Iterator, 794
Observer, 794
Proxy, 794, 797
Singleton, 794, 797

GenericLifeCycleManager interface, 937
GET method, 506-507, 520
GET requests (HTTP), 505
getAttribute() method, 531, 626
getAttributes() method, 108-109, 726-728
getAttributeString() method, 637
getCallerPrincipal() method, 433, 678
getChildNodes() method, 727
getConnection() method, 357, 839
getCreationTime() method, 531
getCurrentValue() method, 629
<getCust> tag, 619-622
getDefaultInstance() method, 468, 494
getDisposition() method, 491
getDocumentElement() method, 727
getEJBHome() method, 433
getEJBLocalHome() method, 433
getElementsByTagName() method, 727
getFileName() method, 491
getFirstChild() method, 727
getFolder() method, 486
getId() method, 530
getInstance() method, 468
getLastAccessedTime() method, 530
getLastChild() method, 727
getLocalHome() method, 303
getLocalTransaction() method, 845
getLocation() method, 452
getLocations() method, 177
getMaxInactiveInterval() method, 531
getMessage() method, 486
getMessages() method, 486
getName() method, 620
getNamespaceURI() method, 726-728
getNodeName() method, 726-728
getNodeType() method, 726-728
getNodeValue() method, 726-728
getObjectInstance() method, 118-119
getParameter() method, 520, 577
getParent() method, 627
getParentNode() method, 727
getPassword Authentication() method, 495
getPreviousSibling() method, 727
getPrimaryKey() method, 253
<getProperty> tag, 579-580
getPropertyNames() method, 408
getRecordFactory() method, 839
getReference() method, 117
getRemoteUser() method, 692
getRequestURI() method, 534
getResourceAsStream() method, 754
getRollbackOnly() method, 344, 437
getSession() method, 531
getSkills() method, 453
getStatus() method, 346
getStore() method, 486
getUserPrincipal() method, 691
getUserTransaction() method, 344-346, 437
getValue() method, 534, 951
global registries, 925
GoF (“Gang of Four” book), 791
granularity, 245, 273
GROUP BY clause (SQL), 984
groups, 664-666
GSSAPI authentication, 696

H

Handle interface, 168
hard-coded properties (JNDI), 90
hasAttributes() method, 728
hasChildNodes() method, 728
hashCode() method, 228
HAVING clause (SQL), 984
HEAD method, 507
<HEAD> tag (HTML), 511
header.jsf file, 814
headers
- Agency Web site, 588
- HTTP (Hypertext Transfer Protocol), 505, 508-509
- JMS (Java Message Service), 408
- multi-part e-mail messages, 479
- SOAP messages, 951-952

Hello service
- HelloServer.java, 886-887
- HelloServerClient.java, 888
- HelloServerPortType.java, 886
- HelloServerSoapBinding Stub.java, 886
- MyHelloService.wsdl, 883-884
- SoapSayHello.java, 889-890

<hello> tag, 605-606, 611-612
HelloServer.java file, 886-887
HelloServerClient.java file, 888
HelloServerPortType.java file, 886
HelloServerSoapBinding Stub.java file, 886
HelloUser.java file, 852
HelloUserClient.java file, 855, 859
HelloUserImpl.java file, 853-854, 858
HelloWorld.java file, 863
HelloWorldImp.c file, 864
hidden form fields, 532
hierarchical tag structures
advertise.jsp page, 632-634
findAncestorWithClass() method, 627
<forEach> tag
code listing, 628-629
TLDs (tag library descriptors), 630
getParent() method, 627
<option> tag, 631
code listing, 629-630
TLDs (tag library descriptors), 630
skills.jsp page, 627-628
Hillside Web site, 792
history
of patterns, 788
of XML (Extensible Markup Language), 703-704
<H*n*> tag (HTML), 511
<home> tag, 145, 185
home interface
EJB (Enterprise JavaBeans) implementation, 140-141, 148, 172
method implementation, 175-176
home methods
Entity EJBs, 229
implementing, 327
LocalHome interface, 310-312
HTML (Hypertext Markup Language), 509
clients
dynamic, 50-51
static, 49-50
compared to XML (Extensible Markup Language), 705
documents
HTML form example, 511-512
simple HTML page example, 510
VerifyForm page, 520
HTML e-mail
absolute URLs, 475
addressing, 473
body text, 473
java.io package, 472
multi-part messages, 476-482
sending, 473-475
tags
nesting, 509
syntax, 509
table of, 510-511
<HTML> tag (HTML), 511
HTTP (Hypertext Transfer Protocol), 57, 504
Basic authentication, 683-685
clients
applet clients, 51
dynamic HTML clients, 50-51
mobile devices, 51
static HTML clients, 49-50
Digest authentication, 683
error handling
client redirection, 529
error-page tag, 528
status codes, 529
HTTPS (HTTP over SSL), 57
methods
DELETE, 507
GET, 506-507
HEAD, 507
OPTIONS, 507
POST, 506
PUT, 507
TRACE, 507
MIME content types, 508
requests
body, 505
GET, 505
headers, 505
parameters, 576-577
request lines, 504
responses, 504-505
body, 505
example, 507
headers, 505, 508-509
response lines, 504
servlet variables, 575
statelessness, 504
status codes
generating, 529
group codes, 508
table of, 508
URIs (Uniform Resource Identifiers), 505-506
HTTPS (Hypertext Transfer Protocol Secure), 659
HttpServletRequest interface, 515
HttpServletResponse interface, 515
HttpSession object, 530-531

HttpSessionActivationListener interface, 542
HttpSessionAttributeListener interface, 542
HttpSessionListener interface, 542
Hypertext Markup Language. *See* HTML
Hypertext Transfer Protocol Secure (HTTPS), 659, 684
Hypertext Transfer Protocol. *See* HTTP
hyphens (-), 710

I

IBM
DeveloperWorks, 792
Web site, 24
Websphere Commerce Business Edition, 24
WSKT Client API, 932-934
WSTK (Web Services Toolkit), 877
ID attribute type, 713, 994
identification variables
from clause, 294
select clause, 295
idioms, 790
IDL (Interface Definition Language), 60-61, 850-851
IDREF attribute type, 713, 994
<if> tag, 645
IgnorableWhitespace() method, 724
IIOP (Internet Inter-ORB Protocol), 850
HelloUserClient.java example, 859
HelloUserImpl.java example, 858
InitialContext objects, 857
Java interpreter properties, 860
PortableRemoteObject interface, 857
rebind() method, 858
RMI-IIOP, 61-62
Transient Name Server, 860
when to use, 865
images, multi-part e-mail messages, 479
IMAP (Internet Message Access Protocol), 464
<IMG> tag (HTML), 511
implementation level (UML), 966
implicit objects (JSPs), 575
#IMPLIED value (DTD attributes), 713, 994
import directive, 572
importing java.io package, 472
in parameter (Class diagrams), 972
INCITs (InterNational Committee for Information Technology Standards) Web site, 977
<<include>> notation (Use Cases), 967
include directive, 570-571
IndexedRecord objects, 840
info directive, 571
init() method, 523, 536
<init-param> tag, 527
initial contexts (JNDI)
creating, 86
InitialContext class, 86, 160, 857
naming exceptions, 86-87
InitialContext class, 86, 160, 857
InitialContext() method, 86
initializing JavaBeans, 581
INNER JOIN statement (SQL), 981-982
inner joins, 981
inout parameter (Class diagrams), 972
input parameters (EJB QL), 298
<INPUT> tag (HTML), 511
INSERT statement (SQL), 980-981
insertBook() method, 837
installing
Agency database, 77-78
J2EE SDK (Software Developers Kit), 30-32
resource adapters, 835-836
integer type, 717
integral option (network security requirements), 689
integrating external resources. *See* external resources
integration-tier patterns
Data Access Object
case study analysis, 804-811
defined, 796
flexibility, 805, 811
maintainability, 805, 811
performance, 811
EJB Observer, 796
Service Activator
availability, 812
case study analysis, 811
defined, 796
performance, 812
scalability, 812
integrity
checksums, 660
defined, 655-656
message digests, 660
InteractionSpec interface, 839
Intercepting Filter pattern, 795-797

Interface Definition Language (IDL), 60-61, 850-851
interfaces
Advertise, 801-802
BodyTag, 608
business, 264
BusinessLifeCycle Manager, 936
BusinessQueryManager, 937
Collection, 287
Concept, 936
Connection, 356-358
ConnectionPoolData Source, 357
defining, 301, 225-230
Document
appendChild() method, 731
cloneNode() method, 731
getAttributes() method, 726-728
getChildNodes() method, 727
getDocumentElement() method, 727
getElementsByTag Name() method, 727
getFirstChild() method, 727
getLastChild() method, 727
getNamespaceURI() method, 726-728
getNodeName() method, 726-728
getNodeType() method, 726-728
getNodeValue() method, 726-728
getParentNode() method, 727
getPreviousSibling() method, 727
hasAttributes() method, 728
hasChildNodes() method, 728
normalize() method, 726
removeChild() method, 731
DocumentBuilderFactory, 725
EJBContext, 168
EJBHome, 167
EJBHomeHandle, 168
EJBLocalHome, 167
EJBLocalObject, 167
EJBObject, 132, 167
EntityBean
implementing, 220, 231-235, 302-305
lifecycle management, 219-224
GenericLifeCycleManager, 937
Handle, 168
HttpServletRequest, 515
HttpServletResponse, 515
HttpSessionActivation Listener, 542
HttpSessionAttribute Listener, 542
HttpSessionListener, 542
InteractionSpec, 839
IterationTag, 622
IterationTag, 608
javax.ejb package, 167
JDO (Java Data Objects), 387-389
JobDAO, 806
local
accessing Entity EJBs, 258-259
implementing methods, 241-243, 312-313
LocalHome, 305-308
finder methods, 308-309
home methods, 310-312
implementing, 235-241
LocalTransaction, 845
Message-driven beans, 439-440
MessageDrivenContext, 433
MessageListener, 414
Organization, 936
PersistenceManager, 385
PortableRemoteObject, 857
remote, 340
RMI remote, 127
SAXParseFactory, 720
Service, 936
ServiceBinding, 936
ServletContext, 524
ServletContextAttribute Listener, 542
ServletContextListener, 542
SessionBean, 168, 175
SingleThreadModel, 570
SQLQueryManager, 937
Status, 346, 360
Tag, 608
XADataSource, 357
InterNational Committee for Information Technology Standards (INCITs) Web site, 977
International Telecommunications Union (ITU) Web site, 660
Internet Explorer, stylesheet support, 747
Internet Inter-Orb Protocol. *See* IIOP
Internet Message Access Protocol (IMAP), 464
invalidate() method, 532, 542
invalidating sessions, 532

invoke method, 891
iPlanet
Application Server Enterprise Edition, 24
Web site, 24
IS NOT NULL operator, 297
IS [NOT] EMPTY operator, 299
IsCallerInRole() method, 433, 678
isEmployee() method, 974
isErrorPage directive, 572
isIdentical() method, 253
isNew() method, 531
isUserInRole() method, 691
isValid() method, 635
IterationTag interface, 608, 622
iterative tags
body content, 622-623
BodyTagSupport class, 622
<forEachJob> tag example, 623-626
IterationTag interface, 622
Iterator pattern, 794
iterator() method, 845
ITU (International Telecommunications Union) Web site, 660

J

J2EE administration tool, 65
J2EE Blueprints, 23-24, 127, 792
J2EE Connector architecture, 62, 829-830
CCI application, 836-837
BookManagerClient.java, 843
BookManagerClient2.java, 848
BookManagerEJB.java, 841-842
BookManagerEJB2.java, 846-847
CceConnectionSpec class, 838
home interface, 837
IndexedRecord objects, 840
InteractionSpec interface, 839
MappedRecord objects, 840
methods, 837-841, 844-846
remote interface, 837
common client interface, 834
connection contract, 831
EIS (Enterprise Information Systems) interaction, 834-835
resource adapter installation, 835-836
roles, 830-831
security contract, 833-834
transaction management, 843
begin() method, 845
BookManagerClient2.java example, 848
BookManagerEJB2.java example, 846-847
contract, 832-833
execute() method, 845
iterator() method, 845
listTitles() method, 844
LocalTransaction interface, 845
rollback() method, 846
Web site, 831
when to use, 865
J2EE HOME environment variable, 152
J2EE RI (Reference Implementation)
deploytool, 405-406, 664
deployment settings, 677-678
EJB (Enterprise JavaBeans), 152-157, 181-182, 251-252, 316
method permissions, 670-674
role mappings, 674-676
transaction configuration, 342
security identity, 669-670
XADataSource interface, 357
J2EE RI for Linux and Unix, 86
J2EE RI for Windows, 85-86
JMS (Java Message Service) implementation, 404
connection factories, 404
destinations, 404
queues, 404-406
many-to-many link tables, 280
J2EE SDK (Software Developers Kit), 28-29. *See also* J2EE RI (Reference Implementation)
Cloudscape
diagnostic messages, 34
starting, 34
troubleshooting, 34-37
documentation, 32
downloading, 30
installing, 30-32
licensing, 29
system requirements, 29-30

J2EE Server tool, 65
j2eeadmin command, 404-405
JAF (JavaBeans Activation Framework), 59
Jakarta Project, 604, 646
jar files, installing classes as, 374
JASB (Java Architecture for XML Binding), 732-733
Java 2 Platform Enterprise Edition Specification, 346
Java applets, 51
Java API for XML Processing (JAXP), 58-59, 718-720
Java API for XML Registries. *See* JAXR
Java APIs for XML Messaging. *See* JAXM
Java APIs for XML-based RPC (JAX-RPC), 25, 876
Java Architecture for XML Binding. (JASB), 732-733
Java Beans. *See* JavaBeans
Java Community Process. *See* JCP
Java Cryptography Extension (JCE), 698
Java Data Objects. *See* JDO
Java Database Connectivity (JDBC), 57, 363-367
Java IDL (Interface Definition Language), 60-61, 851
Java Message Service. *See* JMS
Java Naming and Directory Interface. *See* JNDI
Java Native Interface. *See* JNI
Java Pet Store application, 23-24
Java Remote Method Protocol. *See* JRMP
Java servlets, 47-48, 501
advantages, 47, 502-504
Agency case study, 546-552
class hierarchy, 513
containers (engines), 513
contexts, 524
cookies, 532-534
error handling, 528-530
event listeners, 541-545
filters, 535-541, 544-545
hidden form fields, 532
lifecycle, 522-523
passing parameters to, 519-522
sandboxes, 504, 546
Servlets example, 514-518
sessions, 530-532
single-thread model, 545-546
URL rewriting, 535
Web applications, 525-527
Java Specification Requests (JSRs)
JSR archive, 1002
request process, 1001-1002
Java Transaction API. *See* JTA
Java Transaction Services (JTS), 361-363
Java Web Service (JWS) files
generated WSDL, 904-905
SimpleOrderServer2.jws example, 903-904
java.io package, importing, 472
JavaBeans, 577
creating, 579
declarative security
method permissions, 670-674
role mappings, 674-676
roles, 666-668
security identity, 668-670, 676-678
defined, 578
EJBs (Enterprise JavaBeans). *See* EJBs
parts of, 134
initializing, 581
JAF (JavaBeans Activation Framework), 59
programmatic security
Agency case study, 679-682
getCallerPrincipal() method, 678
isCallerInRole() method, 678
properties
retrieving, 579-580
setting, 580-581
security, 666
JavaBeans Activation Framework (JAF), 59
JavaMail, 60, 461-465
addressing, 473
deleting messages, 489-490
development environment, 465-466
e-mail attachments
creating, 482-483
retrieving, 490-494
sending, 483-485
HTML e-mail
absolute URLs, 475
addressing, 473
body text, 473
java.io package, 472
sending, 473-475
IMAP (Internet Message Access Protocol), 464
mail sessions, 468
message flags, 489

MIME (Multipurpose Internet Mail Extensions), 464-465
multi-part messages
body text, 477
BodyPart objects, 477-478
content type, 479
creating, 476-479
headers, 479
images, 479
MimeMultipart objects, 478-479
sending, 479-482
NNTP (Network News Transport Protocol), 464
plain text messages, 466
addressing, 469
body text, 469
content type, 469
mail environment properties, 467-468
main() method, 467
MimeMessage objects, 468
sending, 470-471
SMTP host access, 467
subjects, 469
POP3 (Post Office Protocol), 463-464
retrieving messages, 485
attachments, 490-494
close() method, 487
connect() method, 486
getFolder() method, 486
getMessage() method, 486
getMessages() method, 486
getStore() method, 486
open() method, 486
RetrieveMail application, 487-488
writeTo() method, 487
SMTP (Simple Mail Transfer Protocol), 463
user authentication
AuthenticateRetrieveMail application, 495-497
Authenticator class, 494
MyAuthenticator class, 494-495
PasswordAuthentication object, 495
JavaServer Pages Standard Tag Library. *See* JSPTL
JavaServer Pages. *See* JSPs
javax.ejb package, 167-168
Entity EJBs (Enterprise JavaBeans), 216-217
EntityBean interface, 231-235, 302-305
exceptions, 179
JAVA_HOME environment variable, 29-30
JAX Pack, 878
JAX-RPC (Java API for XML-based RPC), 25, 876
JAXM (Java API for XML Messaging), 25, 876, 939
clients, 941
configuring, 941-942
message attachments, 951-952
message headers, 951-952
populating messages, 947-951
profiles, 955-956
receiving message with, 959-962
sending message with, 957-959
Providers, 940-941
receiving messages
JAXM profiles, 959-962
simple JAXM client, 952-956
sending messages
JAXM profiles, 957-959
standalone JAXM clients, 942-946
JAXMOrderServer.java file, 953-954
JAXMOrderServiceClient.java file, 943-944
JAXP (Java API for XML Processing), 58-59, 718-720
JAXR (Java API for XML Registries), 25, 876
architecture, 934
client initialization code, 934-935
interfaces
BusinessLifeCycleManager, 936
BusinessQueryManager, 937
Concept, 936
GenericLifeCycleManager, 937
Organization, 936
Service, 936
ServiceBinding, 936
SQLQueryManager, 937
JBoss, 25
JMS (Java Message Service) implementation, 402
connection factories, 403
destinations, 403-404
Web site, 402
JCE (Java Cryptography Extension), 698
JCP (Java Community Process), 999-1000
JSR (Java Specification Request) process, 1001-1002

membership structure
Executive Committees, 1001
Expert Groups, 1000
members, 1000
PMO (Public Management Office), 1001
public involvement, 1000-1001
Web site, 25, 1002
JDBC (Java Database Connectivity), 57, 363-367
JDO (Java Data Objects), 364, 383-384
caches, 384-387
classes and interfaces, 387-389
deployment descriptors, 391
identity, 387
lifecycle, 392
queries, 389-390
SCOs (Second Class Objects), 391
transient transactional objects, 392
JMS (Java Message Service), 60, 395, 430. *See also* J2EE RI
administered objects, 399
clients, 399
connections
closing, 410
defined, 399
domains
defined, 399
point-to-point, 400-402
publish/subscribe, 401, 415-416
goals of, 397-398
JBoss implementation, 402
connection factories, 403
destinations, 403-404
message selectors, 422
messages, 407
body types, 409
consuming, 411-412
creating, 409
defined, 399
headers, 408
persistence, 423
properties, 408-409
sending, 409
multithreading, 425
point-to-point messaging, 396-397
asynchronous messaging, 414-415
closing connections, 410
consuming messages, 411-412
creating messages, 409
message domains, 400-402
message structure, 407-409
PTPSender sample application, 410-411
queues, 406-407
sending messages, 409
synchronous receivers, 412-414
providers, 399
publish/subscribe messaging, 397, 415-416
bulleting board publisher program, 417-418
bulleting board subscriber program, 418-420
durable subscriptions, 420-421
message domains, 401, 415-416
ObjectConnection object, 416
ObjectConnection Factory object, 416
Topic object, 416
TopicPublisher object, 416
TopicSession object, 416
TopicSubscriber object, 416
queues, 404-407
creating with deploytool, 405-406
creating with j2ee admin, 404-405
defined, 399
sessions
acknowledgement modes, 422-423
defined, 399
support for, 398-399
topics, 399
transactions, 423-424
when to use, 396
XA support, 424
JMSCorrelationID field (JMS), 408
JMSDeliveryMode field (JMS), 408
JMSDestination field (JMS), 408
JMSExpiration field (JMS), 408
JMSMessageID field (JMS), 408
JMSPriority field (JMS), 408
JMSRedelivered field (JMS), 408
JMSReplyTo field (JMS), 408
JMSTimestamp field (JMS), 408
JMSType field (JMS), 408

JNDI (Java Naming and Directory Interface), 59, 81, 85
architecture, 83
attributes
defined, 102
modifying, 112-114
reading, 108-109
code bases
defining, 114-117
loading classes from, 114
composite names, 100-101
compound names, 101
configuring
CLASSPATH variable, 85
J2EE RI for Linux and Unix, 86
J2EE RI for Windows, 85-86
server startup, 86
contexts
changing, 94-95
creating, 98-99
destroying, 99-100
directory contexts, 108
initial contexts, 86-87
listing, 96-98
naming exceptions, 86-87
event handling
addNamingListener() method, 120
NamespaceChange Listener event listener, 120
NamingEvent object, 120
NamingListener object, 120-121
ObjectChangeListener event listener, 120
InitialContext, 160
LDAP (Lightweight Directory Access Protocol)
obtaining, 103
OpenLDAP, 104-106
Service Providers, 106-107
testing, 107-108
X.500 names, 102-103
name lookup
JNDILookup.java, 93-94
lookup() method, 93
RMI-IIOP objects, 95-96
sub-contexts, 94-95
naming conventions, 84
objects
binding, 90-92
rebinding, 92
renaming, 93
searching for, 109-111
unbinding, 92-93
properties, 87-88
applet parameters, 90
application properties, 89
hard-coded properties, 90
jndi.properties file, 88-89
references
BookFactory.java application, 118
BookRef.java application, 117-118
getReference() method, 117
JNDIBindBookRef.java application, 119
JNDILookupBookRef.java application, 119-120
Referenceable objects, 117
security, 121-122
LDAP (Lightweight Directory Access Protocol) authentication, 696
properties, 695
SASL (Simple Authentication and Security Layer) authentication, 696-698
special characters, 100
supported naming services, 83-84
URLs (Uniform Resource Locators), 101
jndi.properties file, 88-89, 697-698
JNDIAttributes.java file, 108-109
JNDIBind.java file, 91
JNDIBindBookRef.java file, 119
JNDICodebase.java file, 115-116
JNDICreate.java file, 99
JNDIDestroy.java file, 99-100
JNDIFilter.java file, 111
JNDIListSAMS.java file, 96-97
JNDILookup.java file, 93-94
JNDILookupAny.java file, 107
JNDILookupBook.java file, 116-117
JNDILookupBookRef.java file, 119-120
JNDIModify.java file, 112, 114
JNDISearch.java file, 110
JNDITree.java file, 98

JNI (Java Native Interface), 860-865
HelloWorld.java example, 863
HelloWorldImp.c example, 864
when to use, 865
Job Agency case study. *See* Agency case study
JobDAO interface, 806
jobSummary document
attributes, 708-709
code listing, 708
DTD (document type declaration), 713
namespace, 714-715
XML Schema, 716-717
<jobSummary> tag, 708
JobValueObject object, 806-807
JOD identity, 386
JOIN statement (SQL), 981-982
joins
cross joins, 982
full joins, 983
inner joins, 981
left outer joins, 982
right outer joins, 982
JRMP (Java Remote Method Protocol)
bind() method, 853
HelloUser.java, 852
HelloUserClient.java, 855
HelloUserImpl.java, 853-854
Java interpreter, 856
lookup() method, 854
main() method, 853
rebind() method, 853
RemoteException exceptions, 853
unbind() method, 853
UnicastRemoteObject class, 853
JSP value (<body-content> tag), 608
jspDestroy() method, 569
jspInit() method, 569
JSPs (JavaServer Pages), 46-47, 555-556. *See also* Tag Libraries
actions, 558
Agency case study
advertise.jsp, 592-594, 632-634, 693
agency.jsp, 589-590, 692, 814
agencyName.jsp, 581-582
dateBanner.jsp, 570
errorPage.jsp, 595-597
name.jsp, 572-573
skills.jsp, 627-628
table.jsp, 576-577
tableForm.jsp, 576
updateCustomer.jsp, 594-595
compared to servlets, 600-601
sample Web pages, 556-557
separation of roles, 557
translation and execution, 557
Composite View pattern, 813-817
date.jsp example, 560-561, 563
defined, 556
directives
defined, 558
include, 570-571
page, 571-575
syntax, 570
errors, 563-565
compilation errors, 567-568
HTML presentation errors, 568-569
translation errors, 565-567
implicit objects, 575
JavaBeans, 577
creating, 579
defined, 578
initializing, 581
properties, 579-581
lifecycle
jspDestroy() method, 569
jspInit() method, 569
translation and compilation, 563-565
request parameters, 576-577
running, 557
scripting elements
comments, 560
declarations, 559
defined, 558
expressions, 559
scriptlets, 559-560
separation of concerns, 817-820
servlet variables, 575
structure, 557-558
thread safety, 570
JSPTL (JavaServer Pages Standard Tag Library), 640-641
<choose> tag, 645
downloading, 641
<forEach> tag, 643-644
<forTokens> tag, 645
<if> tag, 645
including in applications, 641-642
scripting language support, 645-646

<jsp:getProperty> tag, 579-580
<jsp:setProperty> tag, 580-581
<jsp:useBean> tag, 579-581
JSRs (Java Specification Requests)
Implementing Enterprise Web Services, 25
J2EE specification, 20
J2EE tools, 65-66
JAX-RPC (Java APIs for XML-based RPC), 25, 876
JAXM (Java APIs for XML Messaging), 25, 876
JAXR (Java APIs for XML Registries), 25, 876
JSR archive, 1002
request process, 1001-1002
JTA (Java Transaction) API, 58, 361
compared to JTS (Java Transaction Services), 362-363
Session beans, 345-349
XA-compliance, 356-358
JTS (Java Transaction Services), 361-363
JWS (Java Web Service) files
generated WSDL, 904-905
SimpleOrderServer2.jws example, 903-904

K-L

Kerberos, 696
key tool, 65
keys
cipher keys, 657
private keys, 658
public keys, 658
labels, formatting, 373
layers (two-tier design), 11-12
layout of Web sites, 586
LDAP (Lightweight Directory Access Protocol), 19, 84
attributes
defined, 102
modifying, 112-114
reading, 108-109
authentication, 696
obtaining, 103
OpenLDAP, 104-106
Service Providers, 106-107
testing, 107-108
X.500 names, 102-103
LEFT JOIN statement (SQL), 982
left outer joins, 982
legacy connectivity, 22
legacy systems, 828-829. *See also* external resources
legal issues, 29
levels (UML), 966
libraries. *See* Tag Libraries
licensing, 29
lifecycle
EJBs (Enterprise JavaBeans)
bean creation restrictions, 143
Entity EJBs, 219-224, 277-279
methods, 138-140
services, 131
stateful Session beans, 193-195, 352-353
stateless Session beans, 168-172
JDO (Java Data Objects), 392
JSPs (JavaServer Pages)
jspDestroy() method, 569
jspInit() method, 569
translation and compilation, 563-565
Message-driven beans, 432-433
servlets, 522-523
tags
doAfterBody() method, 610-611
doEndTag() method, 610
doInitBody() method, 611
doStartTag() method, 610
release() method, 610
lifelines (Sequence diagrams), 973
Lightweight Directory Access Protocol. *See* LDAP
LineItemBean.java file, 914
link tables, 280
Linux
J2EE SDK (Software Developers Kit) installation, 31-32
JNDI (Java Naming and Directory Interface) configuration, 86
list() method, 96-97
listBindings() method, 97-98
listeners
deploying, 543, 545
HttpSessionActivation Listener interface, 542
HttpSessionAttribute Listener interface, 542
HttpSessionListener interface, 542
NamespaceChange Listener, 120
ObjectChangeListener, 120
ServletContextAttribute Listener interface, 542
ServletContextListener interface, 542

listings. *See* code listings
ListSASL.java file, 696-697
listTitles() method, 844
loadDetails() method, 803-804
loading classes from code bases, 114
loadLibrary() method, 862
<local> tag, 185
local interface
Entity EJBs (Enterprise JavaBeans)
accessing, 258-259
BMP (bean-managed persistence) Entity EJBs, 230
CMP (container-managed persistence) Entity EJBs, 301
compared to remote interfaces, 217-219
methods
implementing, 241-243
implementing, 312-313
<local-home> tag, 185
LocalHome interface
BMP (bean-managed persistence) Entity EJBs
custom primary key classes, 227-229
defining, 225
exceptions, 226
finder methods, 226
home methods, 229
CMP (container-managed persistence) Entity EJBs, 301
implementing, 235-241
methods
finder methods, 308-309
home methods, 310-312
implementing, 305-308
locally hosted registries, 929
LocalTransaction interface, 845
log files, 337
<login-config> tag, 685
look and feel (Agency Web site)
agency.css style sheet, 589
footers, 591-592
headers, 588
lookup (JNDI)
JNDILookup.java, 93-94
lookup() method, 93
RMI-IIOP objects, 95-96
sub-contexts, 94-95
<lookup> tag, 615-617
lookup() method, 93-94, 407, 838, 854

M

mail. *See* e-mail
mail.debug property, 468
mail.from property, 468
mail.host property, 468
mail.protocol.host property, 468
mail.protocol.user property, 468
mail.store.protocol property, 467
mail.transport.protocol property, 467
mail.user property, 468
main() method
PTPListener application, 415
RMI over JRMP application, 853
maintaining
patterns
Business Delegate pattern, 821
Composite View pattern, 816
Data Access Object pattern, 805
Service Locator pattern, 822
Session Façade pattern, 799
Value Object pattern, 804
View Helper pattern, 820
state, 905-908
client code, 907-908
deployment descriptors, 906
SessionSimpleOrderServer.java example, 906
many-to-many relationships, 280, 286
many-to-one relationships, 286
MapMessage message type, 409
MappedRecord objects, 840
mapping
serializers, 912-919
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
principals to roles, 674-676
SOAP/WSDL types, 911-912
marketplace registries, 926
marshal() method, 945
MDBPrintMessage bean
code listing, 440
deployment descriptor, 443-445
members (JCP), 1000

membership structure (JCP)
Executive Committees, 1001
Expert Groups, 1000
JCP members, 1000
PMO (Public Management Office), 1001
public involvement, 1000-1001
message acknowledgements
AUTO_ACKNOWLEDGE, 437
DUPS_OK_ ACKNOWLEDGE, 438
message digests, 660
message domains
defined, 399
point-to-point, 400-402
publish/subscribe, 401, 415-416
message persistence, 423
message selectors, 422, 438
<message> tag (WSDL), 885
message-based Web Services, 937. *See also* JAXM (Java API for XML Messaging)
asynchronous services, 939
clients, 938-939
JAXMOrderServer .java, 952-954
JAXMOrderService Client.java, 943-944
ProcessingServlet.java, 960-961
running, 947
SubmittingServlet.java, 957-959
compared to RPC-style services, 937
message attachments, 951-952
message headers, 951-952
populating messages, 947-951
receiving messages
JAXM profiles, 959-962
simple JAXM clients, 952-955
sending messages
JAXM profiles, 957-959
standalone JAXM clients, 942-946
synchronous services, 939
Message-driven beans, 44, 128, 429-430
Agency case study, 447
AgencyBean, 449-450, 735-736
ApplicantMatch bean, 737-739
ApplicationMatch bean, 451-456
deployment, 456-457
MessageSender class, 736-737
queue, 456
RegisterBean, 449-450
sender helper class, 447-449
testing, 457
alternative architectures, 457-458
ApplicationMatch example
code listing, 453-456
deleteByApplicant() method, 452
ejbCreate() method, 451
ejbRemove() method, 451
findByLocation() method, 452
getLocation() method, 452
getSkills() method, 453
InitialContext interface, 451
onMessage() method, 452
skillMatch counter, 453
clients, 430-431
compared to Session and Entity beans, 431
context, 433
creating, 434, 439-440
defined, 430
deploying, 442-457
exception handling, 436
interface implementation, 439-440
interfaces, 431-432
life cycle, 432-433
MDBPrintMessage bean
code listing, 440
deployment descriptor, 443-445
message acknowledgements
AUTO_ ACKNOWLEDGE, 437
DUPS_OK_ ACKNOWLEDGE, 438
message handling, 435-436
message selectors, 438
method-ready pool, 434-435
queues, 441, 456
removing, 435
sender clients, 445-447
close() method, 448
MessageSender helper class, 448-449
MessageSender() method, 447
sendApplicant() method, 448
state, 431-432
testing, 457
transactions, 436-437

MessageDrivenContext interface, 433
MessageListener interface, 414
messages. *See also* e-mail
consuming, 435-436
diagnostic messages
Cloudscape, 34
RI (Reference Implementation), 33-34, 78-79
error messages, 36-37
JMS (Java Message Service, 407
body types, 409
consuming, 411-412
creating, 409
defined, 399
headers, 408
message persistence, 423
message selectors, 422
properties, 408-409
sending, 409
message acknowledgements
AUTO_ACKNOWLEDGE, 437
DUPS_OK_ACKNOWLEDGE, 438
Sequence diagrams, 973
MessageSender class, 736-737
MessageSender helper class, 448-449
MessageSender() method, 447
messaging, 22. *See also* message-based Web Services; Message-driven beans
e-mail, 461-465
attachments, 482-485, 490-494
content type, 479
deleting, 489-490
development environment, 465-466
headers, 479
HTML e-mail, 472-475
images, 479
IMAP (Internet Message Access Protocol), 464
mail sessions, 468
message flags, 489
MIME (Multipurpose Internet Mail Extensions), 464-465
multi-part messages, 476-482
NNTP (Network News Transport Protocol), 464
plain text e-mail, 466-471
POP3 (Post Office Protocol), 463-464
retrieving, 485-488, 490-494
SMTP (Simple Mail Transfer Protocol), 463
user authentication, 494-497
JAXM (Java API for XML Messaging), 25, 939
clients, 941
configuring, 941-942
message attachments, 951-952
message headers, 951-952
populating messages, 947-951
profiles, 955-962
Providers, 940-941
receiving messages, 952-956, 959-962
sending messages, 942-946, 957-959
JAXR (Java APIs for XML Registries), 25
JMS (Java Message Service). *See* JMS
META-INF directory, 525
metadata, 141-142
<method> tag, 340
Method Ready Pool state (Message-driven beans), 432
<method-intf> tag, 671
<method-permissions> tag, 670-671
method-ready pool, 434-435
methods
abstract, 274-275
add(), 287
addAll(), 287
addAttachmentPart(), 951
addBodyElement(), 948
addBodyPart(), 477-478, 482
addChildElement(), 948
addCookie(), 533
addElement(), 736
addNamingListener(), 120
addRecipient(), 469
afterBegin(), 352
appendChild(), 731
applicantXML(), 736
beforeCompletion(), 352
begin(), 346, 845
beginTransactionIf Required(), 349
bind(), 91, 853
BMP Entity EJB interfaces, 225
characters(), 723
cloneNode(), 731
close(), 487, 841
close(), 448
commit(), 346, 423-424
completeTransactionIf Required(), 349
connect(), 486, 494

contextDestroyed(), 542
contextInitialized(), 542
create(), 173
createAttachment(), 951
createCustomer(), 681
createDurableSubscriber(), 420-421
createInteraction(), 839
createJob(), 810-811
createReceiver(), 412
createSubcontext(), 99
createSubscriber(), 420-422
createTextMessage(), 409
ctx.getLocalHome(), 303
DELETE, 507
deleteByApplicant(), 452
deleteCustomer(), 681
destroy(), 523, 537
destroySubcontext(), 99
displayHelloWorld(), 864
doAfterBody(), 610-611, 622-624
doEndTag(), 610
doFilter(), 535-536, 544-545
doGet(), 522, 959
doInitBody(), 611
doPost(), 522, 954
doSelect(), 640
doStartTag(), 610, 621, 624
ejb prefix, 173
ejbActivate(), 172, 198, 221-223, 279, 304
ejbCreate(), 170, 173, 176, 197, 222, 235-237, 278, 305, 433-434, 439, 451, 679
ejbFindByPrimaryKey(), 224-226, 240
ejbHomeDeleteBy Customer(), 303
ejbLoad(), 221, 223, 232, 279, 303
ejbPassivate(), 172, 198, 221-223, 279, 304
ejbPostCreate(), 221-222, 235-237
ejbRemove(), 171, 177, 223, 279, 307, 435, 439, 451
ejbStore(), 221, 223, 232, 279, 303
encodeURL(), 535
endDocument(), 723
endElement(), 721-723
endPrefixMapping(), 723
equals(), 228
error(), 180, 723
examineHeaderElements(), 951
execute(), 840, 845
FatalError(), 723
findAncestorWithClass(), 627
findByLocation(), 452
findByPrimaryKey(), 226, 304
finder methods, 226, 291-293
GET, 506-507, 520
getAttribute(), 531, 626
getAttributes(), 108-109, 726-728
getAttributeString(), 637
getCallerPrincipal(), 433, 678
getChildNodes(), 727
getConnection(), 357, 839
getCreationTime(), 531
getCurrentValue(), 629
getDefaultInstance(), 468, 494
getDisposition(), 491
getDocumentElement(), 727
getEJBHome(), 433
getEJBLocalHome(), 433
getElementsByTagName(), 727
getFileName(), 491
getFirstChild(), 727
getFolder(), 486
getId(), 530
getInstance(), 468
getLastAccessedTime(), 530
getLastChild(), 727
getLocalTransaction(), 845
getLocation(), 452
getLocations(), 177
getMaxInactiveInterval(), 531
getMessage(), 486
getMessages(), 486
getName(), 620
getNamespaceURI(), 726-728
getNodeName(), 726-728
getNodeType(), 726-728
getNodeValue(), 726,-728
getObjectInstance(), 118-119
getParameter(), 520, 577
getParent(), 627
getParentNode(), 727
getPassword Authentication(), 495
getPreviousSibling(), 727
getPrimaryKey(), 253
getPropertyNames(), 408
getRecordFactory(), 839
getReference(), 117
getRemoteUser(), 692
getRequestURI(), 534
getResourceAsStream(), 754
getRollbackOnly(), 344, 437
getSession(), 531
getSkills(), 453
getStatus(), 346
getStore(), 486
getUserPrincipal(), 691
getUserTransaction(), 344-346, 437
getValue(), 534, 951
hasAttributes(), 728
hasChildNodes(), 728

hashCode(), 228
HEAD, 507
IgnorableWhitespace(), 724
init(), 523, 536
InitialContext(), 86
insertBook(), 837
invalidate(), 532, 542
invoke(), 891
IsCallerInRole(), 433, 678
isEmployee(), 974
isIdentical(), 253
isNew(), 531
isUserInRole(), 691
isValid(), 635
iterator(), 845
jspDestroy(), 569
jspInit(), 569
list(), 96-97
listBindings(), 97-98
listTitles(), 844
loadDetails(), 803-804
loadLibrary(), 862
lookup(), 93-94, 407, 838, 854
main()
  PTPListener application, 415
  RMI over JRMP application, 853
marshal(), 945
MessageSender(), 447
MimeBodyPart(), 477
MimeMessage(), 468
ModifyAttributes(), 112
narrow(), 95, 859
newInstance(), 751, 945
newTransformer(), 751
normalize(), 726
notationDecl(), 724
onMessage(), 414-415, 435-436, 452
open(), 486
OPTIONS, 507
outputTable(), 548
parse(), 725
passing by value, 133
permissions, 670-674
POST, 506, 522
processingInstruction(), 724
processPart(), 491
PUT, 507
put(), 467
rebind(), 92, 853, 858
receive(), 412
receiveNoWait(), 413
release(), 610
removeChild(), 731
rename(), 93
resolveEntity(), 724
RMI (Remote Method Invocation), 851-852
  RMI over IIOP, 61-62, 857-860
  RMI over JRMP, 852-857
  when to use, 865
rollback(), 346, 423-424, 846
search(), 109-111
select(), 291-293
send(), 409, 470, 473, 735
sendApplicant(), 448, 735-737
sendRedirect(), 529
setAttribute(), 618
setContent(), 469, 473, 477-479
setDataHandler(), 479
setEntityContext(), 220, 231, 263, 278, 302, 343
setFileName(), 483
setFlag(), 489
setFrom(), 469
setHeader(), 479
setLocation(), 230
setLogin(), 620
setMaintainSession(), 907
setMaxAge(), 534
setMaxInactiveInterval(), 532
setMessageDriven Context(), 433-434, 439
setMessageListener(), 414
setRef(), 625
setRollbackOnly(), 344, 437
setSessionContext(), 170, 343, 838
setSubject(), 469
setText(), 409, 469, 473, 482
setValue(), 534
showResource(), 754
skippedEntity(), 724
start(), 412
startDocument(), 723
startElement(), 720-723
startPrefixMapping(), 723
stored procedure definition, 376
submitOrder(), 905, 908
TRACE, 507
transform(), 751
unbind(), 93, 853
unmarshal, 951
unsetEntityContext(), 232, 263, 278
unsubscribe(), 421
updateDetails(), 255
Warning(), 724
writeFile(), 491
writeTo(), 487, 492, 498
Microsoft .NET framewrk, 26
Microsoft Active Directory, 19
Microsoft Developers Network Web site, 705
MIME (Multipurpose Internet Mail Extensions), 464-465
MimeBodyPart() method, 477
MimeMessage object, 468
MimeMessage() method, 468

MimeMultipart objects, 478-479
mobile devices, 51
modeling application development, 127
modes, session acknowledgement, 422-423
ModificationItem objects, 112
ModifyAttributes() method, 112
modifying attributes
JNDIModify.java application, 112-114
ModificationItem objects, 112
ModifyAttributes() method, 112
modularity, 14-16
monolithic development
disadvantages, 10-11
structure, 10
transitioning to *n*-tier, 26
multi-part messages
body text, 477
BodyPart objects, 477-478
content type, 479
creating, 476-479
headers, 479
images, 479
MimeMultipart objects, 478-479
sending, 479-482
Multi-Schema XML Validator, 705
multimedia e-mail
absolute URLs, 475
addressing, 473
body text, 473
java.io package, 472
multi-part messages
body text, 477
BodyPart objects, 477-478
content type, 479
creating, 476-479
headers, 479
images, 479
MimeMultipart objects, 478-479
sending, 479-482
sending, 473-475
multiplicity element, 317
Multipurpose Internet Mail Extensions (MIME), 464-465
multithreading, 425, 503
MyAuthenticator class, 494-495
MyHelloService.wsdl file, 883-884

N

***n*-tier development, 9-10, 13-14, 28, 38**
advantages, 16-17
business tier
advantages of business components, 39-40
EJBs (Enterprise JavaBeans), 40-43
Entity beans, 43-44
Message-driven beans, 44
Session beans, 43
client tier, 49
applet clients, 51
dynamic HTML clients, 50-51
mobile devices, 51
non-Java clients, 54
peer-to-peer communication, 53
standalone clients, 52-53
static HTML clients, 49-50
Web Services, 54
component frameworks, 15-16
Enterprise Computing Model, 17-18
lifecycle, 18
naming, 18-19
persistence, 18
security, 19-20
transactions, 19
modularity, 14-16
presentation tier, 44-45
development tips, 48-49
JSPs (JavaServer Pages), 46-47
servlets, 47-48
Web-centric components, 45-46
transitioning to, 26
name lookup (JNDI)
JNDILookup.java, 93-94
lookup() method, 93
RMI-IIOP objects, 95-96
sub-contexts, 94-95
name persistence, 92
name() function, 763
<name-from-attribute> tag, 618
<name-given> tag, 618
name.jsp page, 572-573
names, 18-19, 84. *See also* naming services
cmp-fields, 275
cmr-fields, 283
composite names, 100-101
compound names, 101
name lookup (JNDI)
JNDILookup.java, 93-94
lookup() method, 93
RMI-IIOP objects, 95-96
sub-contexts, 94-95
objects, 93

persistence, 92
URLs (Uniform Resource Locators), 101
X.500, 102-103
NamespaceChangeListener event listener, 120
namespaces, 714-715, 991
Naming Service (CORBA), 851
Naming Service (COS), 18
naming services, 131. ***See also*** **JNDI (Java Naming and Directory Interface)**
advantages of, 82-83
CORBA (Common Object Request Broker Architecture), 851
COS (Common Object Services), 18
defined, 82
DNS (Domain Name System), 83
naming conventions, 84
NIS (Network Information Services), 83
support for, 83-84
NamingEvent object, 120
NamingListener object, 120-121
narrow() method, 95, 859
narrowing objects, 95-96
native clients (JMS), 399
NDS (Novell Directory Services), 83
nesting. ***See also*** **hierarchical tag structures**
elements, 707
HTML (Hypertext Markup Language) tags, 509
transactions, 337
.NET framework, 26
Network Information Services (NIS), 83
Network News Transport Protocol (NNTP), 464
Network Security Requirements options, 689-690
New System Variable dialog box, 30
newInstance() method, 751, 945
newline elements, 751
newTransformer() method, 751
NIS (Network Information Services), 83
NMTOKEN attribute type, 713, 994
NMTOKENS attribute type, 713, 994
NNTP (Network News Transport Protocol), 464
node() function, 763
nodes
DOM (Document Object Model)
accessing, 726-728
modifying, 731-732
XML (Extensible Markup Language)
hierarchy, 763
identifying, 762-764
non-Java clients, 54
non-repudiation, 656
none option (network security requirements), 689
NON_PERSISTENT delivery mode, 423
NoPermissionException exception, 112
normalization, 215
normalize() method, 726
NoSuchEntityException exception, 234
NoSuchObjectLocal Exception exception, 253
[NOT] MEMBER OF operator, 300
NOTATION attribute type, 713, 994
notationDecl() method, 724
Novell Directory Services (NDS), 83
NTFS (NT File System), 337
nullable primitives, 298

O

Object Constraint Language (OCL), 973
Object Request Broker (ORB), 850
object-oriented (OO) modeling, 15, 300
object-oriented programming (OOP), 15
ObjectChangeListener event listener, 120
ObjectMessage message type, 409
objects
administered objects, 399
AdvertiseValueObject, 803
Application, 575
binding, 90-91
bind() method, 91
example, 91
name persistence, 92
potential problems, 91-92
BodyPart, 477-478
config, 575
ConnectionFactory, 403
Context. See Contexts, 86
DataSource, 217
HttpSession, 530-531
IndexedRecord, 840
InitialContext, 857

JobValueObject, 806-807
MappedRecord, 840
MimeMessage, 468
MimeMultipart, 478-479
ModificationItem, 112
name lookup
  JNDILookup.java, 93-94
  lookup() method, 93
  RMI-IIOP objects, 95-96
  sub-contexts, 94-95
NamingEvent, 120
NamingListener, 120-121
narrowing, 95-96
out, 575
PageContext, 575
PasswordAuthentication, 495
Queue, 402
QueueBrowser, 402
QueueConnection, 402
QueueConnectionFactory, 402
QueueReceiver, 402
  createReceiver() method, 412
  PTPReceiver example, 413-414
  receive() method, 412
  receiveNoWait() method, 413
  start() method, 412
QueueSender, 402
QueueSession, 402
rebinding, 92
Referenceable, 117
renaming, 93
request, 575
RMIConnectionFactory, 403
RMIXAConnection Factory, 403
searching for
  JNDIFilter.java application, 111
  JNDISearch.java application, 110
  search() method, 109-111
serializable, 133
session, 575
Topic, 416
TopicConnection, 416
TopicConnectionFactory, 416
TopicPublisher, 416
TopicSession, 416
TopicSubscriber, 416
UILConnectionFactory, 403
UILXAConnectionFactory, 403
unbinding, 92-93
XAConnectionFactory, 403
Observer pattern, 794
OCL (Object Constraint Language), 973
OMG
  Enterprise Computing Model, 17-18
    lifecycle, 18
    naming, 18-19
    persistence, 18
    security, 19-20
    transactions, 19
  Web site, 18
one-to-many relationships, 284
onMessage() method, 414-415, 435-436, 452
OO (object-oriented) modeling, 15, 300
OOP (object-oriented programming), 15
open() method, 486
OpenLDAP, 104-106
operations, 971-972
operators
  EJB QL where clause, 297-299
  SQLj Part 2, 380
optimistic transactions, 385
<OPTION> tag (HTML), 511
<option> tag (XML), 631
  code listing, 629-630
  TLDs (tag library descriptors), 630
optional software, 37-38
OPTIONS method, 507
ORB (Object Request Broker), 850
ORDER BY clause (SQL), 984-985
Order.java file, 948-950
Organization interface, 936
out object, 575
out parameter (Class diagrams), 972
outer joins
  full joins, 983
  left outer joins, 982
  right outer joins, 982
outputTable() method, 548

P

<P> tag (HTML), 511
packager, 65
packaging applications, 15, 66-67
  deployment descriptors, 67-68
  EARs (Enterprise Application Archives), 67
  EJB-JAR files, 69
  WAR (Web Archive) files, 70
page directive, 571-575
page scope, 618
Page-by-page Iterator pattern, 796
PageContext object, 575
<param-name> tag, 527

<param-value> tag, 527
parameters
passing to servlets, 519
GET method, 520
getParameter() method, 520
POST method, 522
VerifyData servlet example, 520-521
RMI (Remote Method Invocation) rules, 132
parent axis (XPath), 762
parenthesis (), 993
parse() method, 725
parsing XML (Extensible Markup Language)
DOM (Document Object Model), 725
accessing tree nodes, 726-728
Document interface methods, 725-728, 731-732
DocumentBuilder Factory interface, 725
DOM Parser application, 728-731
modifying tree nodes, 731-732
parse() method, 725
JAXP (Java API for XML Parsing), 58-59
SAX (Simple API for XML), 719-720
DefaultHandler methods, 723-724
endElement() method, 721
SAX Parser application, 721-723
SAXParseFactory interface, 720
startElement() method, 720-721
Partial Value Object pattern, 804
passing parameters
to servlets, 519
GET method, 520
getParameter() method, 520
POST method, 522
VerifyData servlet example, 520-521
by value, 133
passivation
Entity EJBs (Enterprise JavaBeans), 223, 235
stateful Session EJBs (Enterprise JavaBeans), 198-199
PasswordAuthentication object, 495
PATH environment variable, 30-31
paths, search paths, 30
pattern catalogs, 792
pattern languages, 788
patterns, 787-792
advantages of, 790
analysis patterns, 791
applying, 793-794
architectural patterns, 790
Business Delegate
case study analysis, 820-821
defined, 796
maintainability, 821
performance, 821
reliability, 821
case study analysis, 797-798, 822
client-side proxies and delegates, 820-821
data access without EJBs (Enterprise JavaBeans), 804-811
data exchange and Value Objects, 800-804
entity creation, 812-813
JSP (JavaServer Page) creation, 813-817
messages and asynchronous activation, 811
separation of concerns, 817-820
service location, 821-822
Session Facades, 798-800
Command, 205, 794
Composite Entity
case study analysis, 812-813
defined, 796
flexibility, 813
maintainability, 813
performance, 813
Composite View
case study analysis, 813-817
admin.jsp, 815
agency.jsp, 814
flexibility, 816
footer.jsf, 815
header.jsf, 814
maintainability, 816
manageability, 816
performance, 817
defined, 795
Data Access Object
case study analysis, 804-811
defined, 796
flexibility, 805, 811
maintainability, 805, 811
performance, 811
Decorator, 794
defined, 789
design patterns, 790
Dispatcher View, 795
documentation, 789

EJB Observer, 796
ejbLoad() and ejbStore() methods, 323-324
EJBs (Enterprise JavaBeans)
  Adapter classes, 204-205
  business interface, 203
  Entity EJBs, 258
  services, 205
Façade, 794
Fast Lane Reader, 796
finding, 792
Front Controller, 795
future of, 823
history of, 788
idioms, 790
Intercepting Filter, 795, 797
Iterator, 794
Observer, 794
online resources
  Alexander, Christopher, 789-790
  DeveloperWorks, 792
  Hillside, 792
  J2EE Blueprints, 792
  TheServerSide.com, 792
Page-by-page Iterator, 796
pattern catalogs, 792
patterns within patterns, 791
process patterns, 791
Proxy, 794, 797
recommended reading
  Design Patterns - Elements of Reusable Object-Oriented Software, 791
  The Timeless Way of Building, 788
refactoring, 794, 822-823
Service Activator
  availability, 812
  case study analysis, 811
  defined, 796
  performance, 812
  scalability, 812
Service Locator
  availability, 822
  case study analysis, 821-822
  defined, 795
  maintainability, 822
  performance, 821
  reliability, 822
Service to Worker, 795
Session Façade
  case study analysis, 798-800
  defined, 795
  flexibility, 799
  maintainability, 799
  performance, 799
  security, 799
Singleton, 794, 797
structure, 789
Value List Handler, 796
Value Object
  case study analysis, 800-804
  defined, 796
  flexibility, 804
  maintainability, 804
  Partial Value Object, 804
  performance, 804
  scalability, 804
Value Object Builder, 796
View Helper
  case study analysis, 817-820
  defined, 795
  flexibility, 820
  maintainability, 820
  performance, 820
#PCDATA content type, 712, 993
peer-to-peer communication, 53
percent sign (%), 565
performance
  Business Delegate pattern, 821
  Composite View pattern, 817
  EJB (Enterprise JavaBean) containers, 247-248
  Service Activator, 812
  Service Locator pattern, 821
  Session Façade pattern, 799
  Value Object pattern, 804
  View Helper pattern, 820
permissions, 670-674
persistence, 363-365
  bound objects, 92
  EJB (Enterprise JavaBean) services, 132
  Enterprise Computing Model, 18
  JDBC (Java Databasse Connectivity), 365-367
  JDO (Java Data Objects), 383-384
    caches, 384-387
    classes and interfaces, 387-389
    deployment descriptors, 391
    lifecycle, 392
    queries, 389-390
    SCOs (Second Class Objects), 391
    transient transactional objects, 392
  JMS (Java Message Service), 423
  object/relational mapping products, 364
  OODBMSs, 364
  persistent data stores, 336

PowerTier Release 7, 25
SQLj, 367-368
Part 0, 368-373
Part 1, 373-378
Part 2, 378-383
PersistenceManager interface, 385
persistent data stores, 336
PERSISTENT delivery mode, 423
pessimistic locking, 385
Pet Store application, 23-24
pipe character (|), 993
plain text e-mail, 466
addressing, 469
body text, 469
content type, 469
mail environment properties, 467-468
main() message, 467
MimeMessage objects, 468
sending
send() method, 470
SendMail application, 470-471
SMTP host access, 467
subjects, 469
platform independence, 702
platform roles, 62
Application Assemblers, 63
Application Component Providers, 63
Application Deployers, 64
Product Providers, 63
Systems Administrators, 64
Tool Providers, 65
plus sign (+), 993
PMO (Public Management Office), 1001
point-to-point message domains, 400-402
point-to-point messaging, 396-397, 406
asynchronous messaging
exception handling, 415
main() method, 415
MessageListener interface, 414
onMessage() method, 414-415
setMessageListener() method, 414
closing connections, 410
consuming messages, 411-412
creating messages, 409
message domains, 400-402
message structure, 407
body types, 409
headers, 408
properties, 408-409
PTPSender sample application, 410-411
queues, 406-407
sending messages, 409
synchronous receivers
createReceiver() method, 412
PTPReceiver example, 413-414
receive() method, 412
receiveNoWait() method, 413
start() method, 412
POP3 (Post Office Protocol), 463-464
populating messages, 947-951
PortableRemoteObject interface, 857
portal pages, 587
porting EJBs (Enterprise JavaBeans), 181
ports, server port conflicts, 35-36
<portType> tag (WSDL), 885
POST method, 506, 522
Post Office Protocol (POP3), 463-464
PowerTier Release 7, 25
presentation elements, 183-184
presentation errors, 568-569
presentation logic, 12
presentation tier, 44-45
development tips, 48-49
JSPs (JavaServer Pages), 46-47
patterns
Composite View, 795, 813-817
Dispatcher View, 795
Front Controller, 795
Intercepting Filter, 795-797
Service Locator, 795, 821-822
Service to Worker, 795
View Helper, 795, 817-820
separating from business logic, 130-131
servlets, 47-49
Web-centric components, 45-46
presentation-tier patterns
Composite View
case study analysis, 813-817
defined, 795
Dispatcher View, 795
Front Controller, 795
Intercepting Filter, 795-797
Service Locator
availability, 822
case study analysis, 821-822

defined, 795
maintainability, 822
performance, 821
reliability, 822
Service to Worker, 795
View Helper
case study analysis, 817-820
defined, 795
flexibility, 820
maintainability, 820
performance, 820
primary keys
composite, 284
Entity EJBs (Enterprise JavaBeans), 214
CMP (container-managed persistence) Entity EJBs, 272
creating, 222
custom classes, 227-229
field restrictions, 227
immutability, 222
surrogate keys, 243-245
<primkey-field> tag, 308
principal property (JNDI), 121, 695
principals
defined, 661-662
mapping to roles, 674-676
private keys, 658
private registries, 926
process patterns, 791
processingInstruction() method, 724
ProcessingServlet.java file, 960-961
processPart() method, 491
Product Providers, 63
profiles (JAXM), 955-956
receiving messages with, 959-962
sending messages with, 957-959
program development. *See* application development
program listings. *See* code listings
programmatic authorization
Agency case study
advertise.jsp customer name selection, 693
agency.jsp customer options, 692
role references, 693-694
getRemoteUser() method, 692
getUserPrincipal() method, 691
isUserInRole() method, 691
programmatic security, 661
Agency case study, 679-682
EJBs (Enterprise JavaBeans), 678
prologs (XML documents), 990
properties
JavaBean properties
retrieving, 579-580
setting, 580-581
JMS (Java Message Service), 408-409
JNDI (Java Naming and Directory Interface), 87-88
applet parameters, 90
application properties, 89
hard-coded properties, 90
jndi.properties file, 88-89
mail environment properties, 467-468
protocols. *See specific protocol names (for example,* HTTP*)*
Providers (JAXM), 940-941
providers (JMS), 399
proxies
client-side proxies
Business Delegates, 820-821
creating, 885-888
Proxy pattern, 794, 797
PTPReceiver class, 413-414
PTPSender sample application, 410-411
public involvement in (Java Community Process), 1000-1001
public key encryption, 658-659
public keys, 658
Public Management Office (PMO), 1001
public production registries, 929
public test registries, 929
publish/subscribe message domains, 401, 415-416
publish/subscribe messaging, 397, 415-416
bulletin board publisher program, 417-418
bulletin board subscriber program, 418-420
durable subscriptions, 420-421
message domains, 401, 415-416
ObjectConnection object, 416
ObjectConnectionFactory object, 416
Topic object, 416
TopicPublisher object, 416
TopicSession object, 416
TopicSubscriber object, 416

publishers
bulletin board publisher program, 417-418
defined, 397
push messaging model, 396
PUT method, 507
put() method, 467

Q

queries
EJB QL syntax, 293
JDO (Java Data Objects), 389-390
<query> tag, 316
question mark (?), 993
Queue object, 402
QueueBrowser object, 402
QueueConnection object, 402
QueueConnectionFactory object, 402
QueueReceiver object, 402
createReceiver() method, 412
PTPReceiver example, 413-414
receive() method, 412
receiveNoWait() method, 413
start() method, 412
queues
creating, 406-407, 441, 456
with deploytool, 405-406
with j2eeadmin, 404-405
defined, 396, 399
J2EE RI, 404-406
QueueSender object, 402
QueueSession object, 402
quotation marks (“), 101, 990

R

Rational Unified Process (RUP), 966
read-only installation directory, 35
reading attributes, 108-109
realms, 664
realmtool, 65, 664
rebind() method, 92, 853, 858
rebinding objects, 92
receive() method, 412
receiveNoWait() method, 413
receivers (synchronous)
createReceiver() method, 412
PTPReceiver example, 413-414
receive() method, 412
receiveNoWait() method, 413
start() method, 412
receiving messages, 952-956, 959-962
RECENT flag (e-mail), 489
RecipientType class, 498
recommended reading. *See* books
redirecting clients, 529
REF data type, 367, 381
refactoring, 794, 822-823
Reference Implementation. *See* RI
Referenceable objects, 117
references
JNDI (Java Naming and Directory Interface)
BookFactory.java application, 118
BookRef.java application, 117-118
getReference() method, 117
JNDIBindBookRef.java application, 119
JNDILookupBookRef.java application, 119-120
Referenceable objects, 117
role references, 662
referential integrity, 326
refused connections, troubleshooting, 36-37
RegisterBean bean, 449-450
RegisterBusiness.java file, 930-931
registration (EJB services), 131
registries, 923-924
advantages of, 924-925
defined, 924
ebXML R&R (Registry and Repository), 926-927
global registries, 925
JAXR (Java API for XML Registries), 25, 876
architecture, 934
client initialization code, 934-935
interfaces, 936-937
marketplace registries, 926
private registries, 926
searching, 925
site-specific registries, 926
UDDI (Universal Description, Discovery, and Integration)
accessing with JAXR (Java API for XML Registries), 934-937
accessing with UDDI4J, 929-932
accessing with WSKT Client API, 932-934
bindingTemplate structure, 928

locally hosted registries, 929
public production registries, 929
public test registries, 929
service information, 933
tModel structure, 928
relational entities, 215
relationships
cardinality, 281
cascade delete, 312
cascade null, 281
CMP (container-managed persistence) Entity EJBs, 273
cmr-fields, 282-285
manipulating, 286-291
navigability, 282
types, 280-281
composite primary keys, 284
many-to-many, 286
many-to-one, 286
referential integrity, 326
<relationships> tag, 317-320, 322
release() method, 610
reliability
Business Delegate pattern, 821
Service Locator pattern, 822
remote clients, 127
<remote> tag, 145, 185
remote interface, 177-178
remote interfaces
compared to local interfaces, 217-219
Entity EJBs (Enterprise JavaBeans), 283
exporting, 127
stateless Session EJBs (Enterprise JavaBeans), 172
transactions, 340
Remote Method Invocation. *See* RMI
RemoteException exception, 134, 180, 853
remove() method, 150
removeChild() method, 731
rename() method, 93
renaming objects, 93
request object, 575
request scope, 618
request time expressions, 616
requests (HTTP)
body, 505
GET, 505
headers, 505
parameters, 576-577
request lines, 504
#REQUIRED value (DTD attributes), 713, 994
<res-auth> tag, 191
<res-ref-name> tag, 190
<res-sharing> tag, 191
<res-type> element, 190
resolveEntity() method, 724
resource adapters, 835-836
<resource-ref> tag, 145
resource environment references, 192-193
resource managers, 338, 354-356, 359-360
resource names, 145
resource references, 190-192
responses (HTTP)
body, 505
example, 507
headers, 505, 508-509
response lines, 504
<result-type-mapping> element, 316
RetrieveMail application, 487-488
retrieving
EJBs (Enterprise JavaBeans), 149
e-mail messages, 485
attachments, 490-494
close() method, 487
connect() method, 486
getFolder() method, 486
getMessage() method, 486
getMessages() method, 486
getStore() method, 486
open() method, 486
RetrieveMail application, 487-488
writeTo() method, 487
JavaBean properties, 579-580
table data, 981
return types
ejbCreate() method, 226
ejbFind() method, 226
Enumeration, 262
null, 307
RMI (Remote Method Invocation) rules, 132
rewriting URLs (Uniform Resource Locators), 535
RI (Reference Implementation), 32
diagnostic messages, 33-34, 78-79
J2EE RI for Linux and Unix, 86
J2EE RI for Windows, 85-86
security, 663
groups, 664-666
realms, 664
users, 664-666
shutting down, 37
software components, 32
starting, 33-34
troubleshooting, 34
read-only installation directory, 35

refused connections, 36-37
server port conflicts, 35-36
RIGHT JOIN statement (SQL), 982
right outer joins, 982
RMI (Remote Method Invocation), 851-852
EJB services, exporting to remote clients, 127
methods, 132
RMI over IIOP, 61-62, 95-96
HelloUserClient.java example, 859
HelloUserImpl.java example, 858
InitialContext objects, 857
Java interpreter properties, 860
PortableRemoteObject interface, 857
rebind() method, 858
Transient Name Server, 860
RMI over JRMP
bind() method, 853
HelloUser.java, 852
HelloUserClient.java, 855
HelloUserImpl.java, 853-854
Java interpreter, 856
Java interpreter properties, 856
lookup() method, 854
main() method, 853
rebind() method, 853
RemoteException exceptions, 853
unbind() method, 853
UnicastRemoteObject class, 853
when to use, 865
RMIConnectionFactory object, 403
RMIXAConnectionFactory object, 403
roles, 62, 969
Agency case study, 675, 686
Application Assemblers, 63
Application Component Providers, 63
Application Deployers, 64
as security identities, 676-678
Connector architecture, 830-831
creating, 666-668
defined, 662
mapping principals to, 674-676
Product Providers, 63
references, 662
Systems Administrators, 64
Tool Providers, 65
rollback() method, 346, 423-424, 846
<root> elements, 705-706
rows
adding to tables, 980-981
deleting, 980
RPC (Remote Procedure Call) Web Services, 879-881. *See also* SimpleOrderServer
Axis toolkit, 881-883
calling
ServiceClient class, 891
SOAP (Simple Object Access Protocol), 889-891
clients
SimpleOrderClient.java example, 899-900
WSDL (Web Services Description Language) descriptions, 898-899
debugging, 892-894
Hello service. *See* Hello service
implementation requirements, 894
Java proxies, 885-888
JWS (Java Web Service) files
generated WSDL, 904-905
SimpleOrderServer2.jws example, 903-904
service description information, 883
starting with Wsdl2java too deploy.xml example, 902-903
SimpleOrderServer SoapBindingImpl.java example, 902
SimpleOrderServer SoapBindingSkeleton.java example, 901-902
starting with Wsdl2java tool, 900-903
state maintenance, 905-908
client code, 907-908
deployment descriptors, 906
SessionSimpleOrder Server.java example, 906
type mapping
serializers, 912-919
SOAP/WSDL types, 911-912
wrapping existing J2EE components as, 909-911
wrapping Java classes as deployment descriptors, 895-896

deployment information, 897-898
SimpleOrderServer.java example, 894-895
WSDL (Web Services Description Language) documents, 883-885
<run-as> tag, 676
runclient script, 65
running
JAXM (Java API for XML Messaging) clients, 947
JSPs (JavaServer Pages), 557
RUP (Rational Unified Process), 966

S

***Sams Teach Yourself J2EE in 21 Days* CD-ROM**
Agency directory, 76
CaseStudy directory, 76
Examples directory, 76
Exercise directory, 76-77
Solution directory, 76
sandboxes, 504, 546
SASL (Simple Authentication and Security Layer)
jndi.properties file, 697-698
ListSASL.java example, 696-697
SAX (Simple API for XML), 719-720
DefaultHandler methods, 723-724
endElement() method, 721
SAX Parser application, 721-723
SAXParseFactory interface, 720
startElement() method, 720-721
SAXParseFactory interface, 720
scalability
EJB-based applications, 126
Service Activator, 812
servlets, 503
Value Object pattern, 804
Schemas (XML), 715-716, 995-997
Agency case study, 734
example, 716-717
schema type definitions, 717-718
validator, 716
scope, 618
scope attribute (<useBean> tag), 579
<scope> tag, 619
SCOs (Second Class Objects), 391
script variables
adding to page contexts, 618
defining, 637
<getCust> tag example, 619-622
sharing, 626
TLDs (tag library descriptors), 618-619
scripting elements (JSP)
comments, 560
declarations, 559
defined, 558
expressions, 559
scriptlets, 559-560
scripting languages, 645-646
scriptlets, 559-560
scripts
runclient, 65
script variables
adding to page contexts, 618
defining, 637
<getCust> tag example, 619-622
sharing, 626
TLDs (tag library descriptors), 618-619
scripting elements (JSP)
comments, 560
declarations, 559
defined, 558
expressions, 559
scriptlets, 559-560
scripting languages, 645-646
SDK (Software Developers Kit). *See* J2EE SDK
search paths, adding bin directory to, 30
search() method, 109-111
searching
for objects
JNDIFilter.java application, 111
JNDISearch.java application, 110
search() method, 109-111
Web Services, 925
Secure Authentication and Security Layer. *See* SASL
secure authentication schemes, 694
Secure Sockets Layer (SSL), 659
security, 19-20, 22, 653-654
auditing, 656
authentication
Basic authentication, 683-685
client authentication, 655
defined, 654
Digest authentication, 683

Digest MD5, 696
external, 122, 696
forms-based authentication, 683
GSSAPI, 696
HTTPS client authentication, 684
initial identification, 654
JavaMail, 494-497
LDAP (Lightweight Directory Access Protocol), 696
SASL (Simple Authentication and Security Layer), 696-698
secure authentication schemes, 694
user credentials, 655
authorization
declarative authorization, 685-691
defined, 655
programmatic authorization, 691-694
checksums, 660
confidentiality, 655
constraints, 686
creating, 687-689
security-constraint element, 690-691
data integrity, 655-656
declarative, 661
digital certificates, 660-661
EJBs (Enterprise JavaBeans), 131, 666
method permissions, 670-674
programmatic security, 678-682
role mappings, 674-676
roles, 666-668
security identity, 668-670, 676-678
encryption
asymmetric, 658-659
symmetric, 656-658
HTTPS (Hypertext Transfer Protocol Secure), 659
JAAS (Java Authentication and Authorization Service), 58
JCE (Java Cryptography Extension), 698
JNDI (Java Naming and Directory Interface), 121-122, 695
LDAP (Lightweight Directory Access Protocol) authentication, 696
properties, 695
SASL (Simple Authentication and Security Layer) authentication, 696-698
message digests, 660
non-repudiation, 656
principals
defined, 661-662
mapping to roles, 674-676
programmatic, 661
RI (Reference Implementation), 663
groups, 664-666
realms, 664
users, 664-666
roles
Agency case study, 675, 686
as security identities, 676-678
creating, 666-668
defined, 662
mapping principals to, 674-676
references, 662
sandboxes, 546
security constraints, 686
creating, 687-689
<security-constraint> tag, 690-691
security identities
defining, 668-670
roles as, 676-678
Session Façade pattern, 799
SSL (Secure Sockets Layer), 57, 659
Web applications, 682
Basic authentication, 683-685
declarative authorization, 685-691
Digest authentication, 683
forms-based authentication, 683
HTTPS client authentication, 684
programmatic authorization, 691
Agency case study, 692-694
getRemoteUser() method, 692
getUserPrincipal() method, 691
isUserInRole() method, 691
secure authentication schemes, 694
security constraints, 686
creating, 687-689
<security-constraint> tag, 690-691
security contract (Connector architecture), 833-834
security identities
defining, 668-670
roles as, 676-678

<security-constraint> tag, 690-691
<security-identity> tag, 670
SEEN flag (e-mail), 489
select clause (EJB QL), 295-297
select methods
EJB QL, 291-293
finder method alternative, 311
home methods, 327
SELECT statement (SQL), 981
<SELECT> tag, 511, 637-639
selectors, 438
self axis (XPath), 762
self-calls, 975
send() method, 409, 470, 473, 735
sendApplicant() method, 448, 735-737
SendAttachmentMail application, 483-485
sender clients, 445-447
close() method, 448
MessageSender helper class, 448-449
MessageSender() method, 447
sendApplicant() method, 448
SendHTMLMail application, 473-475
sending
e-mail
e-mail attachments, 483-485
HTML e-mail, 473-475
multi-part messages, 479-482
plain text e-mail, 470-471
messages
JAXM (Java API for XML Messaging), 942-946, 957-959
JMS (Java Message Service), 409
SendMail application, 470-471
SendMultiPartMail application, 479-482
sendRedirect() method, 529
separation of concerns (JSPs), 817-820
Sequence diagrams
activations, 973
example, 974-975
lifelines, 973
messages, 973
ser files, 368
serializable objects, 133
serializers, 912-919
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
servers
BEA Weblogic Server, 24
Cloudscape
diagnostic messages, 34
starting, 34
troubleshooting, 34-37
ColdFusion, 24
compatibility, 24-25
EJBs (Enterprise JavaBeans), 126
iPlanet Application Server Enterprise Edition, 24
J2EE RI (Reference Implementation)
diagnostic messages, 33-34, 78-79
starting, 33-34
troubleshooting, 34-37
JBoss, 25
LDAP (Lightweight Directory Access Protocol)
attributes, 102
obtaining, 103
OpenLDAP, 104-106
Service Providers, 106-107
testing, 107-108
X.500 names, 102-103
port conflicts, 35-36
PowerTier Release 7, 25
Transient Name Server, 860
Websphere Commerce Business Edition, 24
XML documents, transforming on, 747
Service Activator pattern
availability, 812
case study analysis, 811
defined, 796
performance, 812
scalability, 812
Service interface, 936
Service Locator pattern
availability, 822
case study analysis, 821-822
defined, 795
maintainability, 822
performance, 821
reliability, 822
Service Providers, 106-107
<service> tag (WSDL), 885
Service to Worker pattern, 795
ServiceBinding interface, 936
ServiceClient class, 891
ServiceDefinition class, 933
ServiceProvider class, 932

ServiceRegistryProxy class, 932
services, 56. ***See also specific service names (for example, NDS)***
connectivity, 22
data access, 22
finding, 821-822
legacy connectivity, 22
servlet engines, 513
<servlet-class> tag, 527
<servlet-mapping> tag, 527
<servlet-name> tag, 527
ServletContext object, 524
ServletContextAttribute Listener interface, 542
ServletContextListener interface, 542
servlets, 47-48, 501. ***See also* JSPs (JavaServer Pages)**
advantages, 47
efficiency, 503
platform independence, 503
scalability, 503
server independence, 503
server interaction, 503-504
tailored for Web applications, 502-503
Agency case study, 546-552
class hierarchy, 513
compared to JSPs (JavaServer Pages), 556, 600-601
sample Web pages, 556-557
separation of roles, 557
translation and execution, 557
compared to Session beans, 171
containers (engines), 513
contexts, 524
cookies, 532-533
creating, 533-534
retrieving information from, 534
error handling
HTTP errors, 528-529
servlet exceptions, 529-530
event listeners, 541
deploying, 543-545
example, 542-543
HttpSessionActivation Listener, 542
HttpSessionAttribute Listener, 542
HttpSessionListener, 542
ServletContext AttributeListener, 542
ServletContextListener, 542
filters
AuditFilter example, 537-538, 544-545
deploying, 538-541
filter chains, 536
methods, 535-537
single-thread model, 545-546
hidden form fields, 532
lifecycle, 522-523
passing parameters to, 519
GET method, 520
getParameter() method, 520
POST method, 522
VerifyData servlet example, 520-521
sandboxes, 504, 546
Servlets example
accessing, 518
code listing, 514-515
deploying, 515-518
HttpServletRequest interface, 515
HttpServletResponse interface, 515
sessions
creating, 531
HttpSession object, 530-531
invalidating, 532
URL rewriting, 535
variables, 575
Web applications
deployment descriptors, 526-527
directory structure, 525-526
when to use, 49
Servlets application
accessing, 518
code listing, 514-515
deploying, 515-518
HttpServletRequest interface, 515
HttpServletResponse interface, 515
session acknowledgement modes, 422-423
session beans, 43
Session EJBs (Enterprise JavaBeans), 128, 165-166
bean-managed transaction demarcation, 345-349
business logic, 212
checklist, 205-206
compared to Message-driven beans, 431
compared to servlets, 171
container-managed transaction demarcation, 392
exceptions, 201-202
javax.ejb package, 167-168
local interfaces, 218
stateful, 166
configuring, 200
lifecycle, 193-195
transactions, 352-353

stateless, 166
changing state, 200
compared to stateful, 201
configuring, 180-181
create() method, 201
defining, 172-175, 196-198
exceptions, 179-180
implementing, 175-178
lifecycle, 168-172
passivation, 198-199
remove() method, 201
timeouts, 199
timeouts, 195
<session> tag, 145, 184-186
Session Façade pattern
case study analysis, 798-800
defined, 795
flexibility, 799
maintainability, 799
performance, 799
security, 799
session object, 575
session scope, 618
<session-config> tag, 527
<session-type> tag, 185
SessionBean interface, 168, 175
SessionContext class, 168
sessions
e-mail sessions, 468
JMS (Java Message Service), 399
QueueSession object, 402
session acknowledgement modes, 422-423
TopicSession object, 416
servlets
creating, 531
HttpSession object, 530-531
invalidating, 532
SessionSimpleOrderServer.java deployment descriptor, 906
SessionSimpleOrderServer.java file, 906
setAttribute() method, 618
setContent() method, 469, 473, 477-479
setDataHandler() method, 479
setEntityContext() method, 220, 231, 263, 278, 302, 343
setFileName() method, 483
setFlag() method, 489
setFrom() method, 469
setHeader() method, 479
setLocation() method, 230
setLogin() method, 620
setMaintainSession() method, 907
setMaxAge() method, 534
setMaxInactiveInterval() method, 532
setMessageDrivenContext() method, 433-434, 439
setMessageListener() method, 414
<setProperty> tag, 580-581
setRef() method, 625
setRollbackOnly() method, 344, 437
setSessionContext() method, 170, 343, 838
setSubject() method, 469
setText() method, 409, 469, 473, 482
setting. *See* configuring
setValue() method, 534
SGML (Standard Generalized Markup Language), 704
sharing script variables, 626
showResource() method, 754
shutting down RI (Reference Implementation), 37
signatures, digital, 658
Simple API for XML. *See* SAX
Simple Mail Transfer Protocol (SMTP), 463
Simple Object Access Protocol. *See* SOAP
simple security (LDAP), 121
simple types, 717
SimpleOrderClient.java file, 899-900
SimpleOrderServer
client, 899-900
deployment descriptor, 895-896
deployment information, 897-898
JWS (Java Web Service) files
generated WSDL, 904-905
SimpleOrderServer2.jws, 903-904
serializers
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
SimpleOrderServer.java file, 894-895
starting with Wsdl2java tool, 900
deploy.xml, 902-903
SimpleOrderServerSoapBindingImpl.java, 902
SimpleOrderServerSoapBindingSkeleton.java, 901-902

state maintenance
client code, 907-908
SessionSimpleOrder Server.java deployment descriptor, 906
SessionSimpleOrderServer.java file, 906
SimpleOrderServer2.jws file, 903-904
SimpleOrderServerSoap BindingImpl.java file, 902
SimpleOrderServerSoap BindingSkeleton.java file, 901-902
single quote (‘), 101, 990
single-thread model, 545-546
single valued path expressions, 296
SingleThreadModel interface, 570
Singleton pattern, 794, 797
site-specific registries, 926
skillMatch counter, 453
skills.jsp page, 627-628
skippedEntity() method, 724
slapadd command, 106
slapcat command, 106
slash (/), 100-101
SMTP (Simple Mail Transfer Protocol), 463
SOAP (Simple Object Access Protocol), 873
calling Web Services with, 889-891
debugging interactions, 892-894
messages
attachments, 951-952
headers, 951-952
populating, 947-951
receiving, 952-956, 959-962
sending, 942-946, 957-959
type mapping, 911-912
SoapSayHello.java file, 889-890
sockets, SSL (Secure Sockets Layer), 57, 659
Software Developers Kit. *See* J2EE SDK
software development. *See* application development
Solaris, J2EE SDK (Software Developers Kit) installation, 31
Solution directory, 76
source code listings. *See* code listings
spaghetti code, 10
specification level (UML), 966
SQL (Structured Query Language), 977-978
ALTER TABLE statement, 978-979
CREATE TABLE statement, 979
CREATE VIEW statement, 979-980
CROSS JOIN statement, 982
DELETE statement, 980
documentation, 977
DROP TABLE statement, 980
DROP VIEW statement, 980
FROM clause, 983
FULL JOIN statement, 983
GROUP BY clause, 984
HAVING clause, 984
INNER JOIN statement, 981-982
INSERT statement, 980-981
JOIN statement, 981-982
LEFT JOIN statement, 982
ORDER BY clause, 984-985
RIGHT JOIN statement, 982
SELECT statement, 981
standards, 977
structured data types, 366
UPDATE statement, 983
WHERE clause, 983-984
SQL3. *See* SQL (Structured Query Language)
SQL99. *See* SQL (Structured Query Language)
SQLj, 363
BMP Entity EJBs, 382-383
overview, 367-368
Part 0, 368-373
Part 1, 373-378
Part 2, 378-383
SQLQueryManager interface, 937
SSL (Secure Sockets Layer), 57, 659
standalone clients, 52-53, 941
Standard Generalized Markup Language (SGML), 704
Standard Tag Library. *See* JSPTL
start() method, 412
startDocument() method, 723
startElement() method, 720-723
starting
Cloudscape, 34
RI (Reference Implementation), 33-34
Web Services, 900-903
startPrefixMapping() method, 723
state
ejbRemove() method, 308
Entity EJBs (Enterprise JavaBeans), 275

Message-driven beans, 431-432
persisting, 259-261, 272, 275
servlets
cookies, 532-534
hidden form fields, 532
sessions, 530-532
URL rewriting, 535
Web Services, 905-908
client code, 907-908
deployment descriptors, 906
SessionSimpleOrder Server.java example, 906
stateful Session EJBs (Enterprise JavaBeans), 166
compared to stateless, 201
lifecycle, 193-195
stateless Session EJBs (Enterprise JavaBeans), 166
compared to stateful, 201
create() method, 201
lifecycle, 168-172
remove() method, 201
statements (SQL)
ALTER TABLE, 978-979
CREATE TABLE, 979
CREATE VIEW, 979-980
CROSS JOIN, 982
DELETE, 980
DROP TABLE, 980
DROP VIEW, 980
FULL JOIN, 983
INNER JOIN, 981-982
INSERT, 980-981
JOIN, 981-982
LEFT JOIN, 982
RIGHT JOIN, 982
SELECT, 981
UPDATE, 983
static member variables, 143
status codes (HTTP)
generating, 529
group codes, 508
table of, 508
Status interface, 346, 360
stored procedures
compared to Session beans, 166
SQLj, 375
StreamMessage message type, 409
string type, 717
<STRONG> tag, 759
Structured Query Language. *See* SQL
stylesheets
default rules, 764-765
validity, 757
XSL (Extensible Stylesheet Language), 744, 997
stylesheets, applying, 746
transformations, 767-778
XSL-FO (Extensible Stylesheet Language-Formatting Objects), 744-745
XSLT (Extensible Stylesheet Transformations), 745-746, 997
compilers, 780-781
default rules, 764-765
elements, 779-780
template rules, 756-761
Transformer class, 751-755
XML nodes, 762-764
subjects (e-mail), 469
submitOrder method, 905, 908
SubmittingServlet.java file, 957-959
subscribers (JMS)
bulletin board subscriber program, 418-420
defined, 397
durable subscriptions, 420-421
Sun Microsystems Multi-Schema XML Validator, 705
surrogate keys, 243-245
symmetric encryption
algorithms, 657-658
Caesar cipher, 656-657
synchronous message-based Web Services, 939
synchronous receivers
createReceiver() method, 412
PTPReceiver example, 413-414
receive() method, 412
receiveNoWait() method, 413
start() method, 412
system exceptions, 179, 254
system-level exceptions, 351
system requirements, 20, 29-30
Systems Administrators, 64
systems design. *See* patterns

T

<TABLE> tag (HTML), 511
table.jsp page, 576-577
tableForm.jsp page, 576
tables
adding rows to, 980-981
creating, 979
deleting rows from, 980
dropping, 980
editing, 978-979

joins
cross joins, 982
full joins, 983
inner joins, 981
left outer joins, 982
right outer joins, 982
retrieving data from, 981
updating, 983
Tag Extra Info (TEI) class, 635-637
Tag interface, 608
Tag Libraries, 603-604
application deployment, 612, 614
Jakarta Project, 604, 646
JSPTL (JavaServer Pages Standard Tag Library), 640-641
choose tag, 645
downloading, 641
<forEach> tag, 643-644
<forTokens> tag, 645
<if> tag, 645
including in applications, 641-642
scripting language support, 645-646
support for, 604
tags, 605
attributes, 615-617, 635
custom tags, 608-609
example, 605-606, 611-612
hierarchical tag structures, 627-634
iterative tags, 622-626
lifecycle, 610-611
script variables, 618-622, 626, 637
tag body processing, 637-640
TLDs (tag library descriptors)
attributes, 616
creating, 606-608
example, 606
file location, 614
<forEach> tag, 630
<option> tag, 630
script variables, 618-619
tag library descriptors. *See* TLDs
<tag> tag, 608
tagdependent value (<bodycontent> tag), 608
<taglib> tag, 607
TagLibs. *See* Tag Libraries
tags
HTML (Hypertext Markup Language)
nesting, 509
syntax, 509
table of, 510-511
JSP (JavaServer Pages)
<jsp:getProperty>, 579-580
<jsp:setProperty>, 580-581
<jsp:useBean>, 579-581
WSDL (Web Services Description Language), 885
XML (Extensible Markup Language), 706. *See also* Tag Libraries
<assembly-descriptor>, 340, 668
<attribute>, 616
attributes, 615-617, 635-637
<body-content>, 608
<cascade-delete>, 317
<choose>, 645
<cmr-field-name>, 318
<cmr-field-type>, 318
content, 712
custom tags, 608-609
<declare>, 619
declaring, 992-993
<display-name>, 527
<!DOCTYPE> tag, 527, 711, 992
element type declarations, 711-712
<error-page>, 527-528
example, 605-606, 611-612
<forEach>, 628-630, 643-644
<forEachJob>, 623-626
<forTokens>, 645
<getCust>, 619-622
<hello>, 605-606, 611-612
hierarchical tag structures, 627-634
<if>, 645
<init-param>, 527
iterative tags, 622-626
<jobSummary>, 708
lifecycle, 610-611
<login-config>, 685
<lookup>, 615-617
<method-intf>, 671
<method-permissions>, 670-671
<name-from-attribute>, 618
<name-given>, 618
nesting, 707
<option>, 629-631
<param-name>, 527
<param-value>, 527
<run-as>, 676
scope, 619

script variables, 618-622, 626
<security-constraint>, 690-691
<security-identity>, 670
<servlet-class>, 527
<servlet-mapping>, 527
<servlet-name>, 527
<session-config?, 527
special characters, 707
structure of, 988-989
<tag>, 608
tag body processing, 637-640
<taglib>, 607
TLDs (tag library descriptors), 606-608, 614-619, 630
<transaction-type>, 437
<variable>, 618-619
<variable-class>, 619
valid elements, 706
XSLT, 779-780
<web-app>, 527

TagSupport class, 609
tcpmon tool, 892, 894
<TD> tag (HTML), 511
TEI (Tag Extra Info) class, 635-637
testing
EJBs (Enterprise JavaBeans), 158-160
LDAP (Lightweight Directory Access Protocol), 107-108
Message-driven beans, 457
text, adding to e-mail, 469, 473
text() function, 763
TextMessage message type, 409
<TH> tag (HTML), 511
TheServerSide.com Web site, 792
thick clients, 129
thin clients, 9, 13
threads
EJBs (Enterprise JavaBeans), 143
JSPs (JavaServer Pages), 570
servlets, 545-546
three-tier development. *See* n-tier development
tiers
business
advantages of business components, 39-40
EJBs (Enterprise JavaBeans), 40-43
entity beans, 43-44
Message-driven beans, 44
session beans, 43
client, 49
applet clients, 51
dynamic HTML clients, 50-51
mobile devices, 51
non-Java clients, 54
peer-to-peer communication, 53
standalone clients, 52-53
static HTML clients, 49-50
Web Services, 54
presentation, 44-45
development tips, 48-49
JSPs (JavaServer Pages), 46-47
servlets, 47-48
Web-centric components, 45-46
***The Timeless Way of Building*, 788**
timeouts
Session EJBs (Enterprise JavaBeans), 195
stateful Session EJBs (Enterprise JavaBeans), 199
TLDs (tag library descriptors)
attributes, 616
creating, 606-608
example, 606
file location, 614
<forEach> tag, 630
<option> tag, 630
script variables, 618-619
tModel structure, 928
Tomcat
Axis toolkit installation, 881-883
installing, 882
Tool Providers, 65
tools
Axis toolkit, 877, 881-883
cleanup tool, 65
Cloudscape Server tool, 65
deployment tool, 65
deploytool, 664
deployment settings, 677-678
method permissions, 670-674
role mappings, 674-676
security identity, 669-670
future of, 65-66
J2EE administration tool, 65
J2EE Server tool, 65
JAX Pack, 878
JSRs (Java Specification Requests), 65-66
key tool, 65
packager, 65

realm tool, 65
realmtool, 664
runclient script, 65
tcpmon, 892, 894
Tool Providers, 65
verifier, 65
Wsdl2java, 900-903
WSTK (Web Services Toolkit), 877
Topic object, 416
TopicConnection object, 416
TopicConnectionFactory object, 416
TopicPublisher object, 416
topics (JMS), 399
TopicSession object, 416
TopicSubscriber object, 416
<TR> tag (HTML), 511
TRACE method, 507
<trans-attribute> tag, 341-342
transaction coordinators, 338
transaction logs, 337
transaction managers, 338, 354-356
<transaction-type> tag, 185, 340, 437
transactions, 19, 22, 131, 336
2PC protocol, 359-360
aborting, 344
bean-managed, 345-349, 436-437
client-demarcated transactions, 350
configuring, 342
connection pooling, 357
container-managed, 338-344, 436-437
deployment descriptors, 340-343
EJBContext, 343
exceptions, 350-351
flat, 337
HTTP (Hypertext Transfer Protocol)
MIME content types, 508
requests, 504-505
responses, 504-509
status codes, 508, 529
J2EE Connector architecture, 843
begin() method, 845
BookManagerClient2.java example, 848
BookManagerEJB2.java example, 846-847
execute() method, 845
iterator() method, 845
listTitles() method, 844
LocalTransaction interface, 845
rollback() method, 846
Java 2 Platform Enterprise Edition Specification, 346
JMS (Java Message Service), 423-424
JTA (Java Transaction API), 58
nested, 337
pessimistic locking, 385
remote interfaces, 340
starting, 346
stateful Session EJBs (Enterprise JavaBeans), 352-353
syntax, 337
transaction management contract, 832-833
XA-compliance, 355-358
transform() method, 751
Transformer class (XSLT), 751-755
Transient Name Server, 860
transient transactional objects, 392
transient variables, 198
translation errors (JSPs), 565-567
transparent persistence, 384
troubleshooting
2PC protocol, 359-360
Cloudscape
diagnostic messages, 34
read-only installation directory, 35
refused connections, 36-37
server port conflicts, 35-36
object binding, 91-92
RI (Reference Implementation)
diagnostic messages, 33-34, 78-79
read-only installation directory, 35
refused connections, 36-37
server port conflicts, 35-36
two phase commit protocol, 354-356, 359-360
two-tier design
disadvantages, 12-13
layers, 11-12
type mapping
serializers, 912-919
BeanOrderServer.java, 915
BeanOrderService client, 916-918
BeanOrderService serializer definition, 915
LineItemBean.java, 914
SOAP/WSDL types, 911-912

U

UDDI (Universal Description, Discovery, and Integration) registries, 873
accessing with JAXR (Java API for XML Registries)
client initialization code, 934-935
interfaces, 936-937
JAXR architecture, 934
accessing with UDDI4J, 929-932
accessing with WSKT Client API, 932-934
bindingTemplate structure, 928
locally hosted registries, 929
public production registries, 929
public test registries, 929
service information, 933
tModel structure, 928
UDDI4J, 929-932
UDDIProxy class, 931
UILConnectionFactory object, 403
UILXAConnectionFactory object, 403
UML (Unified Modeling Language), 965-967
advantages, 966
application development, 127
Class diagrams
associations, 969-970
attributes, 970-971
constraints, 973
generalization, 972
operations, 971-972
conceptual level, 966
implementation level, 966
Sequence diagram
activations, 973
example, 974-975
lifelines, 973
messages, 973
specification level, 966
Use Case diagrams, 967-969
actors, 967
<<extend>> notation, 968
generalization notation, 968
<<include>> notation, 967
notation, 967
Session EJBs, 167
Web site, 966
***UML Distilled, Second Edition*, 966**
unbind() method, 93, 853
unbinding objects, 92-93
UnicastRemoteObject class, 853
Unified Modeling Language. *See* UML
Uniform Resource Locators (URLs), 101, 505-506
absolute URLs, 475
URL rewriting, 535
Universal Description, Discovery, and Integration. *See* UDDI
Unix
Agency database, 277
J2EE RI for Windows, 86
J2EE SDK installation, 31-32
unmarshal() method, 951
unsetEntityContext() method, 232, 263, 278
unsubscribe() method, 421
UPDATE statement (SQL), 983
updateCustomer.jsp page, 594-595
updateDetails() method, 255
updating
Entity EJBs, 223
tables, 983
URLs (Uniform Resource Locators), 101, 505-506
absolute URLs, 475
URL rewriting, 535
Use Case diagrams, 967-969
actors, 967
<<extend>> notation, 968
generalization notation, 968
<<include>> notation, 967
notation, 967
<useBean> tag (JSPs), 579, 581
user authentication
Basic authentication, 683-685
client authentication, 655
defined, 19, 654
Digest authentication, 683
Digest MD5, 696
external, 122, 696
forms-based authentication, 683
GSSAPI, 696
HTTPS client authentication, 684
initial identification, 654
JAAS (Java Authentication and Authorization Service), 58
JavaMail
AuthenticateRetrieveMail application, 495-497
Authenticator class, 494
MyAuthenticator class, 494-495
PasswordAuthentication object, 495

LDAP (Lightweight Directory Access Protocol), 696
SASL (Simple Authentication and Security Layer)
jndi.properties file, 697-698
ListSASL.java example, 696-697
secure authentication schemes, 694
user credentials, 655
user authorization
declarative authorization
network security requirements, 689-690
roles, 685
security constraints, 686-691
defined, 19, 655
JAAS (Java Authentication and Authorization Service), 58
programmatic authorization
Agency case study, 692-694
getRemoteUser() method, 692
getUserPrincipal() method, 691
isUserInRole() method, 691
user credentials, 655
USER flag (e-mail), 489
utilities. *See* tools

V

valid XML (Extensible Markup Language) documents, 704
validating
attributes
example, 635-637
isValid() method, 635
XML (Extensible Markup Language) documents, 705
Value List Handler pattern, 796
Value Object Builder pattern, 796
Value Object pattern
case study analysis, 800-804
Advertise interface, 801-802
AdvertiseValueObject object, 803
loadDetails method, 803-804
defined, 796
flexibility, 804
maintainability, 804
Partial Value Object, 804
performance, 804
scalability, 804
<variable> tag, 618-619
<variable-class> tag, 619
variables
environment variables
CLASSPATH, 31, 85
JAVA_HOME, 29-30
PATH, 30-31
identification variables
from clause, 294
select clause, 295
script variables
adding to page contexts, 618
defining, 637
<getCust> tag example, 619-622
sharing, 626
TLDs (tag library descriptors), 618-619
servlet variables, 575
static variables, 143
transient variables, 198
verifier, 65
VerifyData servlet
parameters, 521
VerifyForm HTML page, 520
VerifyForm page, 520
View Helper pattern
case study analysis, 817-820
AgencyBean use, 818-819
AgencyBean.java, 817-818
tag library, 819-820
defined, 795
flexibility, 820
maintainability, 820
performance, 820
views
creating, 979-980
dropping, 980

W

W3C Web site, 997
WAR (Web Archive) files, 70, 909
Warning() method, 724
Web applications
deployment descriptors, 526-527
directory structure, 525-526
EJBs (Enterprise JavaBeans), 128
security, 682
Basic authentication, 683-685
declarative authorization, 685-691

Digest authentication, 683
forms-based authentication, 683
HTTPS client authentication, 684
programmatic authorization, 691-694
secure authentication schemes, 694
Web Archive (WAR) files, 70, 909
Web authentication
Basic authentication, 683-685
Digest authentication, 683
forms-based authentication, 683
HTTPS client authentication, 684
secure authentication schemes, 694
Web authorization
declarative authorization
network security requirements, 689-690
roles, 685
security constraints, 686-691
programmatic authorization
Agency case study, 692-694
getRemoteUser() method, 692
getUserPrincipal() method, 691
isUserInRole() method, 691
Web browsers, 743
Web components, 21
Web interface (Agency case study)
advertise.jsp page, 592-594, 632-634, 693
agency.css style sheet, 589
agency.jsp page, 589-590, 692, 814
agency.ldif configuration file, 105-106
AgencyBean.java, 582-584
agencyName.jsp page, 581-582
dateBanner.jsp page, 570
deploying, 597-600
EJB references, 598
errorPage.jsp, 595-597
look and feel, 588-592
agency.css style sheet, 589
footers, 591-592
headers, 588
name.jsp page, 572-573
portal page, 587
skills.jsp, 627-628
structure and navigation, 585-587
table.jsp page, 576-577
tableForm.jsp page, 576
updateCustomer.jsp, 594-595
Web Service Flow Language (WSFL), 873
Web Service registries, 923-924
advantages of, 924-925
defined, 924
ebXML R&R (Registry and Repository), 926-927
global registries, 925
marketplace registries, 926
private registries, 926
searching, 925
site-specific registries, 926
UDDI (Universal Description, Discovery, and Interaction), 928
accessing with JAXR (Java API for XML Registries), 934-937
accessing with UDDI4J, 929-932
accessing with WSKT Client API, 932-934
bindingTemplate structure, 928
locally hosted registries, 929
public production registries, 929
public test registries, 929
service information, 933
tModel structure, 928
Web Services, 54, 869-870. *See also* Web Service registries
advantages of, 872
architecture
customer/service interaction, 873-874
service implementations, 875
defined, 870-872
J2EE Web Services
architecture, 875-876
integrating with existing components, 878-879
JSRs (Java Specification Requests), 876-877
toolkits, 877-878
message-based, 937. *See also* JAXM (Java API for XML Messaging)
asynchronous services, 939
clients, 938-939
compared to RPC-style services, 937

message attachments, 951-952
message headers, 951-952
populating messages, 947-951
receiving messages, 952-955, 959-962
sending messages, 942-946, 957-959
synchronous services, 939
protocols
ebXML (Electronic Business XML), 873
SOAP (Simple Object Access Protocol), 873
UDDI (Universal Description, Discovery, and Integration), 873
WSDL (Web Services Description Language), 873, 883-885
WSFL (Web Service Flow Language), 873
RPC-style Web Services, 879-881. *See also* SimpleOrderServer
Axis toolkit, 881-883
calling, 889-891
clients, 898-900
debugging, 892-894
Hello service. *See* Hello service
implementation requirements, 894
Java proxies, 885-888
JWS (Java Web Service) files, 903-905
service description information, 883
starting with Wsdl2java tool, 900-903
state maintenance, 905-908
wrapping existing J2EE components as, 909-911
wrapping Java classes as, 894-898
WSDL (Web Services Description Language) documents, 883-885
type mapping
serializers, 912-919
SOAP/WSDL types, 911-912
Web sites, 870
Web Services Description Language (WSDL), 873
MyHelloService.wsdl, 883-885
type mapping, 911-912
Web Services Toolkit (WSTK), 877
Web sites
Active Directory, 19
Alexander, Christopher, 789-790
Apache Jakarta Project, 604, 646
Axis toolkit, 877
BEA Weblogic Server, 24
ColdFusion, 24
Connector architecture, 831
DeveloperWorks, 792
DNS (Domain Name System), 19
ebXML (Electronic Business XML), 873
EJB 2.0 specification, 143
Hillside, 792
IBM, 24
INCITS (InterNational Committee for Information Technology Standards), 977
iPlanet, 24
ITU (International Telecommunications Union), 660
J2EE Blueprints, 23, 792
J2EE compatibility suite, 24
J2EE SDK (Software Developers Kit), 30
J2EE specification, 20
Jakarta Project, 646
JBoss, 25, 402
JCP (Java Community Process), 25, 1002
JDBC 3.0, 366
JSR (Java Specification Requests) archive, 1002
LDAP (Lightweight Directory Access Protocol), 19
Microsoft Developers Network, 705
Multi-Schema XML Validator, 705
OOMG, 18
Persistence, 25
RUP (Rational Unified Process), 966
SOAP (Simple Object Access Protocol), 873
TheServerSide.com, 792
UDDI (Universal Description, Discovery, and Integration), 873
UML (Unified Modeling Language), 966
W3C, 997
WSDL (Web Services Description Language), 873

WSFL (Web Service Flow Language), 873
X.500, 19
XALAN, 748
<web-app> tag, 527
Web-centric components, 45-46
WEB-INF directory, 525
Weblogic Server, 24
Websphere Commerce Business Edition, 24
well-formed XML (Extensible Markup Language) documents, 704, 708
where clause (EJB QL), 297-300
WHERE clause (SQL), 983-984
whitespace, 767-769
wildcards, 298
Windows
Agency database, 277
J2EE RI for Windows, 85-86
J2EE SDK installation, 31
wizards, 166
wrapping
J2EE components as Web Services, 909-911
Java classes as Web Services
deployment descriptors, 895-896
deployment information, 897-898
SimpleOrderServer.java example, 894-895
writeFile() method, 491
writeTo() method, 487, 492, 498
WSDL (Web Services Description Language), 873
MyHelloService.wsdl example, 883-885
type mapping, 911-912
Wsdl2java tool, starting Web Services with, 900-903
deploy.xml example, 902-903
SimpleOrderServerSoapBindingImpl.java example, 902
SimpleOrderServerSoapBindingSkeleton.java example, 901-902
WSFL (Web Service Flow Language), 873
WSTK (Web Services Toolkit), 877, 932-934

X-Z

X.500 protocol, 19, 102-103
X/Open XA, 424
XA-compliance, 355-358, 424
XAConnectionFactory object, 403
XADataSource interface, 357
XALAN
command line operation, 750
configuring, 748-749
newline elements, 751
XML documents, transforming, 749
XML (Extensible Markup Language) documents, 701-702, 987-988. *See also* tags
advantages, 703
attributes, 708-709, 712-713
case sensitivity, 988
comments, 709-710, 990
compared to HTML, 705
declarations, 706
defined, 988
deployment descriptors, 67-68
DTDs (document type declarations), 710-711, 989-990
attributes, 712-713
defined, 706
element content, 712
element type declarations, 711-712
example, 713-714
DTDs (Document Type Definitions), 992-995
attribute declarations, 993-994
deployment descriptors, 182, 249
element declarations, 992-993
EJB (Enterprise JavaBeans) references, 188-189
entity references, 994-995
environment entries, 187-188
example, 992
resource environment references, 192-193
resource references, 190-192
presentation elements, 183-184
Session element, 184-186
enforcing structure of, 991
DTDs (Document Type Definitions), 992-995
XML Schemas, 995-997
history of, 703-704

JASB (Java Architecture for XML Binding), 732-733
JAXM (Java APIs for XML Messaging), 25
JAXP (Java API for XML Parsing), 58-59, 718-720
JAXR (Java APIs for XML Registries), 25
jobSummary document
  attributes, 708-709
  code listing, 708
  DTD (document type declaration), 713
  namespace, 714-715
  XML Schema, 716-717
namespaces, 714-715, 991
online documentation, 997
parsing with DOM (Document Object Model)
  accessing tree nodes, 726-728
  Document interface methods, 725-728, 731-732
  DocumentBuilder Factory interface, 725
  DOM Parser application, 728-731
  modifying tree nodes, 731-732
  parse() method, 725
parsing with SAX (Simple API for XML), 719-720
  DefaultHandler methods, 723-724
  endElement() method, 721
  SAX Parser application, 721-723
  SAXParseFactory interface, 720
  startElement() method, 720-721
platform-independent data exchange, 702-703
prologs, 990
root elements, 705-706
special characters, 990-991
support for, 22
tags
  <assembly-descriptor>, 340, 668
  <attribute>, 616
  attributes, 615-617, 635-637
  <body-content>, 608
  <cascade-delete>, 317
  <choose>, 645
  <cmr-field-name>, 318
  <cmr-field-type>, 318
  content, 712
  custom tags, 608-609
  <declare>, 619
  declaring, 992-993
  <display-name>, 527
  <!DOCTYPE> tag, 527, 711, 992
  element type declarations, 711-712
  <error-page>, 527-528
  example, 605-606, 611-612
  <forEach>, 628-630, 643-644
  <forEachJob>, 623-626
  <forTokens>, 645
  <getCust>, 619-622
  <hello>, 605-606, 611-612
  hierarchical tag structures, 627-634
  <if>, 645
  <init-param>, 527
  iterative tags, 622-626
  <jobSummary>, 708
  lifecycle, 610-611
  <login-config>, 685
  <lookup>, 615-617
  <method-intf>, 671
  <method-permissions>, 670-671
  <name-from-attribute>, 618
  <name-given>, 618
  nesting, 707
  <option>, 629-631
  <param-name>, 527
  <param-value>, 527
  <run-as>, 676
  scope, 619
  script variables, 618-622, 626
  <security-constraint>, 690-691
  <security-identity>, 670
  <servlet-class>, 527
  <servlet-mapping>, 527
  <servlet-name>, 527
  <session-config>, 527
  special characters, 707
  structure of, 988-989
  <tag>, 608
  tag body processing, 637-640
  <taglib>, 607
  TLDs (tag library descriptors), 606-608, 614-619, 630
  <transaction-type>, 437
  <variable>, 618-619
  <variable-class>, 619
  valid elements, 706
  XSLT, 779-780
  <web-app>, 527
transformations, 742, 746-747
  adding comments, 769-770
  attribute values, 770-771
  compilers, 780-781

creating elements, 771-774
defining attributes, 774-776
elements, 779-780
numbering elements, 777-778
whitespace, 767-769
valid documents, 704
validating, 705
well-formed documents, 704, 708
XML Schemas, 715-716, 995-997
Agency case study, 734
example, 716-717
schema type definitions, 717-718
validator, 716
XPath, 762-764, 997-998
XPointer, 762, 997
XML-RPC, 25
XPath, 762-764, 997-998
XPointer, 762, 997
XSL (Extensible Stylesheet Language), 744, 997
elements, 765-769, 775-780
stylesheets, applying, 746
transformations
adding comments, 769-770
attribute values, 770-771
creating elements, 771-774
defining attributes, 774-776
numbering elements, 777-778
whitespace, 767-769
XSL-FO (Extensible Stylesheet Language-Formatting Objects), 744-745
XSLT (Extensible Stylesheet Transformations), 745-746, 997
compilers, 780-781
elements, 779-780
stylesheets, 755
default rules, 764-765
template rules, 756-761
Transformer class, 751-755
XML nodes, 762-764

JAVA™ 2 SOFTWARE DEVELOPMENT KIT STANDARD EDITION VERSION 1.3 SUPPLEMENTAL LICENSE TERMS

These supplemental license terms ("Supplemental Terms") add to or modify the terms of the Binary Code License Agreement (collectively, the "Agreement"). Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Agreement, or in any license contained within the Software.

1. **Internal Use and Development License Grant**. Subject to the terms and conditions of this Agreement, including, but not limited to, Section 2 (Redistributables) and Section 4 (Java Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce the Software for internal use only for the sole purpose of development of your Java™applet and application ("Program"), provided that you do not redistribute the Software in whole or in part, either separately or included with any Program.
2.. **Redistributables**. In addition to the license granted in Paragraph 1above, Sun grants you a nonexclusive, non-. transferable, limited license to reproduce and distribute, only as part of your separate copy of JAVA™ 2 RUNTIME ENVIRONMENT STANDARD EDITION VERSION 1.3 software, those files specifically identified as redistributable in the JAVA™ 2 RUNTIME ENVIRONMENT STANDARD EDITION VERSION 1.3 "README" file (the "Redistributables") provided that: (a) you distribute the Redistributables complete and unmodified (unless otherwise specified in the applicable README file), and only bundled as part of the JavaTM applets and applications that you develop (the "Programs:); (b) you do not distribute additional software intended to supersede any component(s) of the Redistributables; (c) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables; (d) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, and (e) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.
3. **Separate Distribution License Required**. You understand and agree that you must first obtain a separate license from Sun prior to reproducing or modifying any portion of the Software other than as provided with respect to Redistributables in Paragraph 2 above.

4. **Java Technology Restrictions**. You may not modify the Java Platform Interface ("JPI", identified as classes contained within the "java" package or any subpackages of the "java" package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI. In the event that you create an additional class and associated API(s) which (i) extends the functionality of a Java environment, and (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API, you must promptly publish broadly an accurate specification for such API for free use by all developers. You may not create, or authorize your licensees to create additional classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any class file naming convention. Refer to the appropriate version of the Java Runtime Environment binary code license (currently located at http://www.java.sun.com/jdk/index.html) for the availability of runtime code which may be distributed with Java applets and applications.
5. **Trademarks and Logos**. You acknowledge and agree as between you and Sun that Sun owns the Java trademark and all Java.related trademarks, service marks, logos and other brand designations including the Coffee Cup logo and Duke logo ("Java Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/trademarks. Any use you make of the Java Marks inures to Sun's benefit.
6. **Source Code**. Software may contain source code that is provided solely for reference purposes pursuant to the terms of this Agreement.
7. **Termination**. Sun may terminate this Agreement immediately should any Software become, or in Sun's opinion be likely to become, the subject of a claim of infringement of a patent, trade secret, copyright or other intellectual property right.

JAVA™ DEVELOPMENT TOOLS FORTE™ FOR JAVA™, RELEASE 3.0, COMMUNITY EDITION SUPPLEMENTAL LICENSE TERMS

These supplemental license terms ("Supplemental Terms") add to or modify the terms of the Binary Code License Agreement (collectively, the "Agreement"). Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Agreement, or in any license contained within the Software.

1. **Software Internal Use and Development License Grant**. Subject to the terms and conditions of this Agreement, including, but not limited to Section 3 (Java(TM) Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce internally and use internally the binary form of the Software complete and unmodified for the sole purpose of designing, developing and testing your [Java applets and] applications intended to run on the Java platform ("Programs").
2. **License to Distribute Redistributables**. In addition to the license granted in Section 1 (Redistributables Internal Use and Development License Grant) of these Supplemental Terms, subject to the terms and conditions of this Agreement, including but not limited to Section 3 (Java Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified (unless otherwise specified in the applicable README file), and only bundled as part of your Programs, (ii) you do not distribute additional software intended to supersede any component(s) of the Redistributables, (iii) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (iv) for a particular version of the Java platform, any executable output generated by a compiler that is contained in the Software must (a) only be compiled from source code that conforms to the corresponding version of the OEM Java Language Specification; (b) be in the class file format defined by the corresponding version of the OEM Java Virtual Machine Specification; and (c) execute properly on a reference runtime, as specified by Sun, associated with such version of the Java platform, (v) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, and (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.
3. **Java Technology Restrictions**. You may not modify the Java Platform Interface ("JPI", identified as classes contained within the "java" package or any subpackages of the "java" package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI. In the event that you create an additional class and associated API(s) which (i) extends the functionality of the Java platform, and (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API,

you must promptly publish broadly an accurate specification for such API for free use by all developers. You may not create, or authorize your licensees to create, additional classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

4. **Java Runtime Availability**. Refer to the appropriate version of the Java Runtime Environment binary code license (currently located at http://www.java.sun.com/jdk/index.html) for the availability of runtime code which may be distributed with Java applets and applications.

5. **Trademarks and Logos**. You acknowledge and agree as between you and Sun that Sun owns the SUN, SOLARIS, JAVA, JINI, FORTE, STAROFFICE, STARPORTAL and iPLANET trademarks and all SUN, SOLARIS, JAVA, JINI, FORTE, STAROFFICE, STARPORTAL and iPLANET.related trademarks, service marks, logos and other brand designations ("Sun Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/trademarks. Any use you make of the Sun Marks inures to Sun's benefit.

6. **Source Code**. Software may contain source code that is provided solely for reference purposes pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement.

7. **Termination for Infringement**. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right.

For inquiries please contact: Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303

Hey, you've got enough worries.

Don't let IT training be one of them.

Get on the fast track to IT training at InformIT, your total Information Technology training network.

InformIT | www.informit.com | SAMS

■ Hundreds of timely articles on dozens of topics ■ Discounts on IT books from all our publishing partners, including Sams Publishing ■ Free, unabridged books from the InformIT Free Library ■ "Expert Q&A"—our live, online chat with IT experts ■ Faster, easier certification and training from our Web- or classroom-based training programs ■ Current IT news ■ Software downloads ■ Career-enhancing resources

InformIT is a registered trademark of Pearson. Copyright ©2001 by Pearson.
Copyright ©2001 by Sams Publishing.

Other Related Titles

Java P2P Unleashed
Robert Flenner, Michael Abbott, Toufic Boubez, Navaneeth Krishnan, Rajam Ramamurti, Frank Sommers
0-672-32399-0
49.99 6/14/2002

Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI
Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama
0-672-32181-5
49.99 12/12/2001

Sams Teach Yourself Wireless Java with J2ME in 21 Days
Michael Morrison
0-672-32142-4
39.99 6/27/2001

Java Connector Architecture: Building Enterprise Adaptors
Atul Apte
0-672-32310-9
49.99 5/6/2002

Jini and JavaSpaces Application Development
Robert Flenner
0-672-32258-7
49.99 12/5/2001

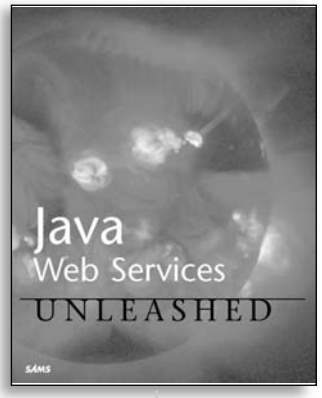
JMX: Managing J2EE with Java Management Extensions
Marc Fleury, Juha Lindfors, The Jboss Group
0-672-32288-9
39.99 1/31/2002

JBoss Administration and Development
Scott Stark, Marc Fluery
0672323478
49.99 3/28/2002

Enhydra XMLC Java Presentation Development
David H. Young
0-672-32211-0
39.99 1/15/2002

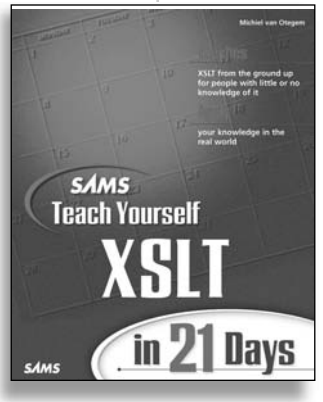
SAMS

www.samspublishing.com



Java Web Services Unleashed
Robert Brunner, Frank Cohen, Francisco Curbera, Darren Govoni, Steven Haines, Matthias Kloppmann, Benoît Marchal, K. Scott Morrison, Arthur Ryman, Joseph Weber, Mark Wutka
0-672-32363-X
49.99 4/19/2002

JXTA: Java P2P Programming
Daniel Brookshier, Darren Govoni, Navaneeth Krishnan, Juan Carlos Soto
0-672-32366-4
39.99 3/22/2002



Sams Teach Yourself XSLT in 21 Days
Michael van Otegem
0-672-32318-4
39.99 2/2002

All prices are subject to change.

What's on the CD-ROM

The companion CD-ROM contains Sun Microsystem's Java Software Development Kit (SDK) version 1.3, Forte 3.0 Community Edition, JBoss, BEA's WebLogic Server, and more software tools plus the source code from the book.

Windows Installation Instructions

1. Insert the disc into your CD-ROM drive.
2. From the Windows desktop, double-click on the My Computer icon.
3. Double-click on the icon representing your CD-ROM drive.
4. Double-click on the icon titled `START.EXE` to run the installation program.
5. Follow the on-screen prompts to finish the installation.

If you have the AutoPlay feature enabled, the `START.EXE` program starts automatically whenever you insert the disc into your CD-ROM drive.

Linux, Mac OS X, and UNIX Installation Instructions

1. Insert the disc into your CD-ROM drive.
2. If you have a volume manager on your UNIX workstation, the disc will be automatically mounted. If you do not have a volume manager, you need to manually mount the CD-ROM. For example, if you were mounting the CD-ROM on a Linux workstation, you would type `mount -tiso9660 /dev/cdrom /mnt/cdrom`
3. Follow the instructions in `readme.html` or `readme.txt` to install the software components.

The mount point on your UNIX workstation must exist before mounting the CD-ROM to it. The mount point in the example is the usual mount point for a CD-ROM, but you can use any existing directory as a mount point. If you are having difficulty or insufficient permission rights to mount a CD-ROM, please review the man page for mount or talk to your system administrator.

Use of this software is subject to the Sun Microsystems, Inc. Binary Code License Agreement contained on page ___ of the accompanying book. Read this agreement carefully. By opening this package, you are agreeing to be bound by the terms and conditions of this agreement.

By opening this package, you are also agreeing to be bound by the following agreement:

You may not copy or redistribute the entire CD-ROM as a whole, Copying and redistribution of individual software programs on the CD-ROM is governed by terms set by individual copyright holders.

The installer and code from the author(s) are copyrighted by the publisher and the author(s). Individual programs and other items on the CD-ROM are copyrighted or are under an Open Source license by their various authors or other copyright holders.

This software is sold as-is without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither the publisher nor its dealers or distributors assumes any liability for any alleged or actual damages arising from the use of this program. (Some states do not allow for the exclusion of implied warranties, so the exclusion may not apply to you.)

NOTE: This CD-ROM uses long and mixed-case filenames requiring the use of a protected-mode CD-ROM Driver.